Roberto Flórez Rueda

Algoritmos, estructuras de datos y programación orientada a objetos

ECOE EDICIONES

Roberto Flórez Rueda
Profesor Asociado
Universidad de Antioquia

Algoritmos, estructuras de datos y programación orientada a objetos

Flórez Rueda, Roberto
Algoritmos, estructura de datos y programación orientada a objetos / Roberto Flórez Rueda. — Bogotá : Ecoe Ediciones, 2005.
376 p. ; 24 cm. — (Area informática)
ISBN 958-648-394-0
1. Estructuras de datos (Computadores) 2. Programación Orientada a objetos (Computación) 3 Algoritmos I. Tít. II. Serie.
005.73 cd 19 ed.
AJA9829

CEP-Banco de la República-Biblioteca Luis-Angel Arango

Colección: Textos universitarios
Área: Informática
Primera edición: Bogotá, D.C., febrero de 2005
ISBN: 958-648-394-0

E-mail: rflorez@udea.edu.co

E-mail: correo@ecoeediciones.com
www.ecoeediciones.com
Calle 32 bis No. 17-22, Pbx. 2889821, Fax.3201377

Coordinación editorial: Adriana Gutiérrez M.
Autoedición: Magda Rocío Barrero
Portada: Patricia Díaz
Fotomecánica: Imagen Gráfica
Impresión: Esfera Editores Ltda.
Calle 63 bis No. 70-45, Tel. 2242061
Impreso y hecho en Colombia

A mis padres y hermanos,
a mi esposa e hijo
porque me quieren mucho

Agradecimientos
a Dios
porque me lo ha dado todo

Tabla de contenido

Figuras

Tablas

Introducción

El presente texto es el resultado de un poco más de 15 años impartiendo los cursos de Algoritmos II y Estructuras de datos I en la Universidad de Antioquia.

Los temas tratados y los algoritmos desarrollados son independientes del lenguaje de programación que utilicen los alumnos.

Los algoritmos se presentan en un seudocódigo, el cual es un acomodo de los lenguajes Pascal y C++.

A continuación presento el seudocódigo que utilizo.

Para la instrucción de asignación utilizo el símbolo igual "=". Ejemplo:

```
a = 3.14 * radio ^ 2
```

A la variable **a** se le asigna el resultado de multiplicar 3.14 por el valor almacenado en la variable **radio** elevada al cuadrado. Los símbolos para las expresiones aritméticas son: + para la suma, - para la resta, * para la multiplicación, / para la división y ^ para la potenciación. Los símbolos para las operaciones de comparación son: > para mayor que, >= para mayor o igual que, < para menor que, <= para menor o igual que, = para igual y <> para diferente. Los símbolos para los operadores lógicos son: **and** para la conjunción, **or** para la disyunción y **not** para la negación.

Las instrucciones de decisión son el **if** y los **casos**.

La forma general para la instrucción **if** es:

```
if  <condición> then
        <instrucciones>
else
        <instrucciones>
end(if)
```

Las formas generales para la instrucción **casos** son:

Forma 1:

```
casos de <expresión>
        :valor 1:
                <instrucciones>
        :valor 2:
                <instrucciones>
            .
            .
            .
        :valor n:
                <instrucciones>
fin(casos)
```

Forma 2:

```
casos
        :<condición 1>:
                <instrucciones>
        :<condición 2>:
                <instrucciones>
                .
                .
                .
        :<condición n>:
                <instrucciones>
fin(casos)
```

Las instrucciones de ciclos son: **while**, **for** y **do**. Las formas generales son:

Instrucción **while:**

```
while <condición> do
    <instrucciones>
end(while)
```

Instrucción **for**:

```
for variable = valor inicial [to o downto] valor final do
        <instrucciones>
end(for)
```

Cuando se utilice **to**, la variable se incrementará de a uno, y cuando se utilice **downto** la variable se decrementará de a uno.

Instrucción **do**

```
do
      <instrucciones>
while <condición>
```

La instrucción de lectura es **read**, cuya forma general es:

read(lista de variables separadas por comas)

La instrucción de escritura es **write**, cuyas formas generales son:

write('mensaje') o
write(lista de variables separadas por comas)

1

SISTEMAS NUMÉRICOS PESADOS

1.1 DEFINICIÓN Y CONCEPTOS

Sistemas numéricos pesados son sistemas numéricos en los cuales cada dígito con los que se representan cantidades tiene dos valores: uno intrínseco (por el hecho de ser ese dígito) y otro posicional (por el sitio que ocupe en la hilera de dígitos).

En general, una hilera de dígitos para representar cantidades se presenta así:

$$D_n D_{n-1} \ldots D_2 D_1 D_0 \qquad \{1\}$$

en la cual, al dígito menos significativo se le asigna la posición cero (D_0) y a partir de ahí, hacia la izquierda las posiciones serán 1,2,... hasta el dígito de la posición **n**.

Si queremos representar cantidades con parte entera y parte decimal la hilera es:

$$D_n D_{n-1} \ldots D_2 D_1 D_0 D_{-1} D_{-2} D_{-3} \ldots D_{-m}$$

Observe que a los dígitos correspondientes a la parte decimal se les asignan las posiciones –1, –2, –3, ... y así sucesivamente.

Para determinar el valor representado en una hilera de dígitos se procede así:

$$\text{Valor} = D_n * b^n + D_{n-1} * b^{n-1} + \ldots + D_1 * b^1 + D_\varnothing * b^0 + \cdot D_{-1} * b^{-1} + D_{-2} * b^{-2} + \ldots + D_{-m} * b^{-m}$$

Siendo ***b*** la base utilizada para representar cantidades.

En general: $Valor = \sum_{i=-m}^{i=n} D_i * b^i$

Ej. Si se tiene el número $(316.25)_{10}$ en base diez, el valor representado será:

$$3*10^2 + 1*10^1 + 6*10^0 + 2*10^{-1} + 5*10^{-2}$$

Nosotros estamos familiarizados con la representación de cantidades en **base diez**.

La **base** se define de acuerdo al número de dígitos que se utilizan para representar cantidades. Nuestra base se denomina base diez porque utiliza diez dígitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Sin embargo, existen tantas bases como se quiera, para representar cantidades:

Base 4, utiliza cuatro dígitos:	(0, 1, 2, 3)
Base 2, utiliza dos dígitos:	(0, 1)
Base 8, utiliza ocho dígitos:	(0, 1, 2, 3, 4, 5, 6, 7)
Base 16, utiliza dieciseis dígitos:	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

Para indicar en cuál base se halla una hilera de dígitos escribiremos la hilera de dígitos entre paréntesis y en la parte externa derecha, como subíndice, la base.

$$(D_n D_{n-1} \ldots D_2 D_1 D_0 D_{-1} D_{-2} D_{-3} \ldots D_{-m})_b$$

Es bueno observar cómo se generan las hileras de dígitos para representar cantidades. Si estamos en base diez, comenzamos desde el 0 hasta el 9; cuando se llega al 9, como se agotaron los dígitos, se añade un uno (1) a la izquierda (10) y se comienzan a generar nuevamente los dígitos desde el cero hasta el nueve en la posición de la derecha: 11, 12, 13, 14, 15,... hasta 19; nuevamente se han agotado los dígitos en la posición de la derecha, entonces se suma 1 al dígito de la izquierda y se generan nuevamente los dígitos del 0 al 9 en la posición de la derecha: 20, 21, 22, 23,...,29; y así sucesivamente hasta llegar a 99. En este punto se han agotado los dígitos en ambas posiciones, entonces se añade un nuevo 1 a la izquierda y se generan nuevamente todos los dígitos de las dos posiciones de la derecha: 100, 101, 102,..., 199 y así sucesivamente.

Si queremos generar hileras en base cinco, cuyos dígitos son 0, 1, 2, 3, 4 sería: 0, 1, 2, 3, 4; se han agotado los dígitos de la base cinco, entonces continuamos con 10, 11, 12, 13, 14; luego 20, 21, 22, 23, 24.

Por consiguiente, el 5 de la base diez, en base cinco, es el 10; el 6 de la base diez, en base cinco, es el 11; el 7 de la base diez, en base cinco, es el 12, y así sucesivamente. El 10, de la base diez, en base cinco es el 20.

En este punto estamos interesados en convertir cantidades representadas en una base hacia otra.

Plantearemos los siguientes casos:

- Conversión de un número en base ***b*** hacia base diez.
- Conversión de un número en base diez hacia base ***b***.
- Conversión de números cuyas bases son potencias entre sí.

1.2 CONVERSIÓN DE UN NÚMERO EN BASE *b* HACIA BASE DIEZ

Si se quiere convertir el $(314)_5$ hacia base diez procedemos así:

$3*5^2+1*5^1+4*5^0$ y efectuamos las operaciones en base diez

$3*25+1*5+4*1=84$ por consiguiente $(314)_5=(84)_{10}$

Si se tiene $(275.25)_8$ y se desea trasladar a base 10

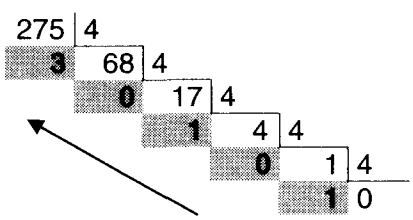
$$2*8^2+7*8^1+5*8^0+2*8^{-1}+5*8^{-2}$$
$$128 + 56 + 5 + 0.25 + 0.078125 = 189.32812$$

es decir, $(275.25)_8=(189.32812)_{10}$

1.3 CONVERSIÓN DE UN NÚMERO EN BASE DIEZ HACIA BASE *b*

Para convertir un número en base diez hacia base ***b*** hay que considerar por separado la parte entera y la parte decimal. Ilustremos la conversión con un ejemplo: Convertir $(275.125)_{10}$ hacia base 4.

La parte entera se convierte mediante divisiones sucesivas por la base destino, hasta obtener cociente cero.



Por cada división se señala el residuo correspondiente.

El número en base ***b*** será la concatenación de los residuos en orden inverso de aparición. Por consiguiente

$$(275)_{10} = (10103)_4$$

La parte decimal se convierte mediante multiplicaciones sucesivas de la parte decimal, por la base destino hasta obtener 0 en la parte decimal o alcanzar la precisión deseada.

$$0.125 * 4 = 0.5$$
$$0.5 * 4 = 2.0$$

Por cada multiplicación se señala la parte entera obtenida, y la parte decimal resultante se multiplica nuevamente por la base destino. Las partes enteras de cada multiplicación conformarán el número en la base **b** en el mismo orden de aparición.

$$(0.125)_{10} = (0.02)_4$$

1.4 CONVERSIÓN DE NÚMEROS CUYAS BASES SON POTENCIAS ENTRE SÍ

Consideremos las bases $\mathbf{b_1}$ y $\mathbf{b_2}$ y llamemos **k** la relación de potencias entre ellas, es decir:

$$\mathbf{b_1 = (b_2)^k}$$

Entonces, cada dígito de la base $\mathbf{b_1}$ requiere **k** dígitos de la base $\mathbf{b_2}$.

Como ejemplo tomemos las bases dos y ocho, en donde $8 = 2^3$, por consiguiente, cada dígito de la base ocho requiere tres dígitos de la base dos.

Construyamos entonces la siguiente tabla:

Tabla 1.1

Base 8	base 2
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

y si deseamos convertir un número en base ocho hacia la base dos basta con utilizar dicha tabla reemplazando cada dígito de la base ocho por sus correspondientes tres dígitos de la base dos.

Ejemplo: convertir $(607514)_8$ hacia base 2

El 6 es el 110, el 0 es 000, el 7 es 111, el 5 es 101, y así sucesivamente, por consiguiente el número en base dos es: $(110000111101001100)_2$.

Si deseamos hacer el proceso inverso, es decir, convertir desde base dos hacia base ocho, conformaremos grupos de tres dígitos, partiendo desde el extremo derecho del número en base dos, y cada grupo de tres dígitos se reemplaza por su correspondiente dígito de la base ocho, según la tabla.

Ejemplo: convertir $(1100010101)_2$ hacia base 8.

Conformando grupos de a tres dígitos de derecha a izquierda tenemos:
101, 010, 100 y 1. El último grupo que es sólo un 1 lo completamos con dos ceros a la izquierda, ya que ceros a la izquierda en parte entera no afectan el valor representado. Por consiguiente, utilizando la tabla 1, el número en base 8 es: $(1425)_8$.

En caso de que el número a convertir tenga parte decimal, los dígitos a continuación del punto se agrupan también de a tres dígitos, pero de izquierda a derecha.

Ejemplo: convertir $(1100111.1101)_2$ a base ocho

Agrupando la parte entera de derecha a izquierda tendremos: 111, 100 y 001.

Agrupando la parte decimal de izquierda a derecha tendremos: 110 y 100. El último grupo se completa con ceros a la derecha ya que ceros a la derecha en parte decimal no afecta el valor representado.

Por consiguiente, utilizando la tabla 1.1:

$$(1100111.1101)_2 = (147.64)_8$$

EJERCICIOS PROPUESTOS

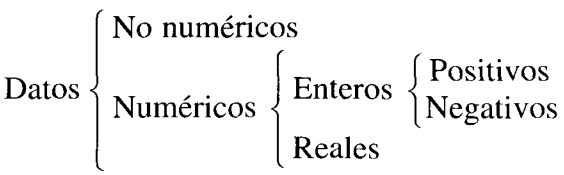
Efectúe las siguientes conversiones de bases:

1. $(DEA)_{16}$ a base 8
2. $(3.14)_{7}$ a base 12
3. $(100111.001)_{2}$ a base 16
4. $(21120212.200112)_{3}$ a base 9
5. $(675.316)_{8}$ a base 4

2

REPRESENTACIÓN DE DATOS DENTRO DE UN COMPUTADOR

Con el fin de representar datos dentro de un computador éstos se clasifican entre *no numéricos* y *numéricos*, los datos numéricos a su vez se clasifican entre enteros y reales, y los enteros se clasifican entre positivos y negativos.



2.1 DATOS NO NUMÉRICOS

La unidad básica para representar datos dentro de un computador es el bit, el cual es un elemento biestable. Por convención, se dice que cada bit representa el cero o el uno.

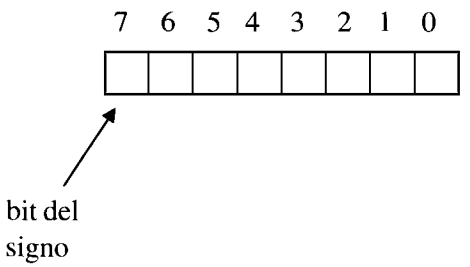
Con el fin de representar los diferentes caracteres se han construido grupos de 8 bits, los cuales se denominan bytes, y cada byte, de acuerdo a la combinación de unos y ceros que tenga, representa un carácter diferente.

Esta convención es la que se conoce como código **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). De los ocho bits que tiene un byte el código ASCII sólo se utilizan 7 para representar datos, el octavo bit se utiliza para control de transmisión de datos.

De acuerdo con lo anterior, si deseamos representar la palabra 'palabra' en memoria, ésta requerirá 7 bytes, uno por cada carácter.

2.2 DATOS NUMÉRICOS ENTEROS

Para representar datos numéricos el computador utiliza otra agrupación de bits, la cual va desde 8 hasta 64 bits, dependiendo de la arquitectura de cada máquina. Nosotros, por simplicidad, consideraremos grupos de 8 bits, cuya forma general es:



En general la representación de datos numéricos entero se hace de tres formas:

- Signo-magnitud.
- Signo-complemento a uno.
- Signo-complemento a dos.

La estructura general de estos ocho bits es:

Bit del signo: 0 significa que el número es positivo, 1 que es negativo.

Los otros bits representan un número decimal codificado en binario.

Por consiguiente, si tenemos la siguiente hilera 01101110 el número decimal representado en ella es: +110 en base diez.

Los números positivos se representan sólo en signo-magnitud.

Los enteros negativos se pueden representar de las tres formas vistas anteriormente.

El complemento a uno de un número binario se obtiene cambiando ceros por unos y unos por ceros. Si deseamos obtener el complemento a uno del número binario 0110010, éste será: 1001101.

El complemento a dos de un número binario se obtiene sumándole uno al complemento a uno, por consiguiente si deseamos obtener el complemento a dos del número binario 0110010, éste será 1001110.

Es decir, si queremos representar el –110 de la base diez lo podremos hacer de tres formas:

11101110 signo-magnitud
10010001 signo-complemento a uno
10010010 signo-complemento a dos

Nuestros ejercicios consistirán en determinar cuál es el valor almacenado en alguna estructura de esas de acuerdo a la representación dada, y el viceversa, es decir, representar un entero en una estructura dada.

Consideremos algunos ejemplos:

1. Determine el valor almacenado en la siguiente estructura, la cual se halla en la representación signo-magnitud: **01000100**

Como el primer bit es cero el número es positivo, y el valor almacenado en ella se obtiene convirtiendo el número binario 1000100 a base diez, el cual es 68.

Por consiguiente **el valor almacenado en la estructura 01000100 en la representación signo-magnitud es + 68**.

2. Determine el valor almacenado en la siguiente estructura, la cual se halla en la representación signo-complemento a uno: **11000100**

Como el primer bit es uno el número es negativo, y el valor almacenado se obtiene determinando el complemento a uno de los otros 7 bits y luego convirtiendo este resultado a base diez. El complemento a uno de 1000100 es 0111011, el cual en base diez es 59. Por consiguiente **el valor almacenado en 11000100 en la representación signo-complemento a uno es –59**

3. Determine el valor almacenado en la siguiente estructura, la cual se halla en la representación signo-complemento a dos: **11000100**.

Como el primer bit es uno el número es negativo, y el valor almacenado se obtiene hallando el complemento a dos de los otros 7 bits y luego convirtiendo este resultado a base diez. El complemento a dos de 1000100 es 0111100, cuyo valor

en base diez es 60. Por consiguiente **el valor almacenado en 11000100 en la representación signo-complemento a dos es –60.**

4. Representar el – 43 en las tres formas:

a. En signo-magnitud: primero convertimos el 43 a base 2, el cual es 0101011, por consiguiente la representación en signo magnitud es **10101011**.

b. En signo-complemento a uno: convertimos el 43 a base 2, el cual es 0101011, luego le hallamos su complemento a uno, el cual es 1010100, por consiguiente la representación en signo-complemento a uno es **11010100**.

c. En signo-complemento a dos: Convertimos el 43 a base 2, el cual es 0101011, luego le hallamos su complemento a dos el cual es 1010101, por consiguiente la representación en signo-complemento a dos es **11010101**.

En general, digamos que los enteros positivos se representan en la forma signo-magnitud, mientras que los enteros negativos se representan en la forma signo-complemento a dos.

Veamos ahora cuáles son las ventajas de tener dicha representación. Si queremos sumar dos enteros representados en la forma signo-magnitud uno de los cuales es positivo y el otro negativo, debemos determinar cuál de los dos números es mayor en valor absoluto, luego, del número mayor restar el menor y asignarle al resultado el signo del número mayor en valor absoluto.

Si tenemos representados los positivos en la forma signo-magnitud y los negativos en la forma signo-complemento a dos bastará con sumar los números dígito a dígito, incluyendo el bit del signo y obtenemos el resultado correcto: si en el resultado el bit del signo es 0 el resultado es un número positivo y está en la forma signo-magnitud; si en el resultado el bit del signo es 1 el resultado es un número negativo y está en la forma signo-complemento a dos.

Como ejemplo consideremos los números +9, -9, +16 y –16.

El + 9 es 00001001
El – 9 es 11110111
El + 16 es 00010000
El – 16 es 11110000

La suma de estos números, con base en la representación dada es:

(9)	(00001001)	(9)	(00001001)
+(16)	(00010000)	+(-16)	(11110000)
(25)	(00011001)	(-7)	(11111001)
(-9)	(11110111)	(-9)	(11110111)
+(-16)	(11110000)	+(16)	(00010000)
(-25)	(11100111)	(7)	(00000111)

2.3 DATOS NUMÉRICOS REALES

Los números reales los representaremos en estructuras de 32 bits de dos formas: una que utiliza exponentes en base 2 y otra que utiliza exponentes en base 16.

Estructura de 32 bits que utiliza exponentes en base 2

La distribución de los 32 bits se presentan en la figura 2.1.

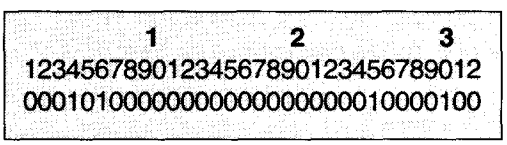


Figura 2.1

- El primer bit es el bit del signo: 0 significa que el número es positivo, 1 que el número es negativo.
- Del bit 2 al 24 es un número binario conocido como mantisa.
- Del bit 25 al 32 es un número decimal representado en binario conocido como característica.

Nuestros objetivos son nuevamente dos: uno, dada una estructura de estas, determinar el valor representado en ella, y dos, dado un número real representarlo en dicha estructura.

Ocupémonos del primero de ellos.

Tomaremos como ejemplo la estructura de la figura 2.1.

Para determinar el valor almacenado en una estructura de estas seguimos los siguientes pasos:

1. Determinar el signo del número. En nuestro ejemplo el primer bit es cero, por consiguiente el número es positivo.
2. Determinar el valor de la característica. Convertir el número de base 2 a base diez:
   $(10000100)_2 = (132)_{10}$
   característica = 132
3. Determinar el valor del exponente:
   Exponente = característica – 128
   Exponente = 132 – 128 = 4
4. Valor almacenado = signo(.mantisa)*2^e
   En nuestro ejemplo
   Valor almacenado = +(.00101000000000000000000)*2^4
   Valor almacenado = $+(0010.10)_2$
5. Convertir el valor almacenado a base diez.
   Valor almacenado es +2.5

Veamos ahora el proceso inverso: representar el -65.125 de la base diez en estructura de 32 bits que utiliza exponentes en base 2.

1. convertir el 65.125 a base 2.
   $(65.125)_{10} = (1000001.001)_2$
2. Determinar valor del exponente. Este se determina desplazando el punto de tal manera que el número quede de la forma **.1**α siendo α una hilera cualquiera de unos y ceros. Si el punto hay que desplazarlo hacia la izquierda el valor del exponente es positivo, de lo contrario es negativo. En nuestro ejemplo el punto hay que desplazarlo 7 posiciones hacia la izquierda, por tanto el valor del exponente es +7.
3. Calcular la característica:
   Característica = 128 + exponente
   Característica = 128 + 7 = 135
4. Convertir la característica a base 2
   $(135)_{10} = (10000111)_2$
5. Escribir el número:

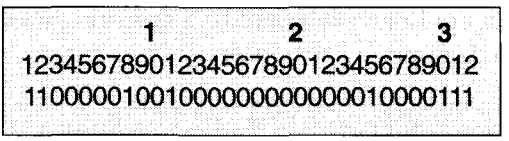


Figura 2.2

Estructura de 32 bits que utiliza exponentes en base dieciséis

La distribución de los 32 bits se presenta en la figura 2.3.

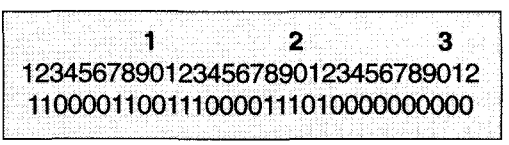


Figura 2.3

- El primer bit es para el signo: 0 positivo, 1 negativo
- Del bit dos al 8 es la característica, la cual es número decimal representado en binario.
- Del bit 9 al 32 es la mantisa, la cual es un número hexadecimal representado en binario

El valor almacenado será: signo(.mantisa)*16^e

Como ejemplo consideremos la estructura de la figura 2.3.

1. Determinar el valor de la característica.
   Característica = $(1000011)_2 = (67)_{10}$
2. Determinar el valor del exponente
   Exponente = característica – 64
   Exponente = 67 – 64 = 3
3. Convertir la mantisa a base 16
   $(00111000011101000000000)_2 = (3872)_{16}$
4. Valor almacenado = $-(.3872)_{16} * 16^3$
   Valor almacenado = $-(387.2)_{16}$
5. Convertir valor almacenado en base 16 hacia base diez.
   $-(387.2)_{16} = -(896.125)_{10}$

El proceso viceversa consiste en dado un número en base diez, representarlo en estructura de 32 bits que utiliza exponentes en base dieciseis. Ilustremos este

proceso con un ejemplo: Representar el 316.5 en estructura de 32 bits que utiliza exponentes en base dieciseis.

1. Convertir 316.5 a base 16: $(316.5)_{10} = (13C.8)_{16}$
2. Mover el punto hacia el extremo izquierdo para determinar el valor del exponente: $(13C.8) = (.13C8)*16^3$, por consiguiente el valor del exponente es 3.
3. Calcular la característica:
   Característica = 64 + exponente
   Característica = 64 + 3 = 67
4. Convertir la característica a base 2:
   $(67)_{10} = (1000011)_2$
5. Convertir la mantisa de base 16 a base 2:
   $(.13C8)_{16} = (0001001111001000)_2$
6. Construir la estructura:

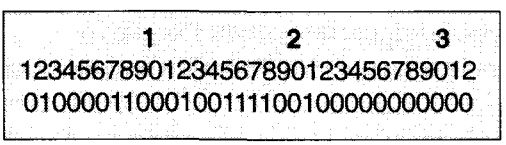


Figura 2.4

EJERCICIOS PROPUESTOS

1. Represente en estructura de 32 bits que utilice exponentes en base 2:
   - $(3.14)_{10}$
   - $(- .032675)_{10}$
   - $(41252)_{10}$

2. Represente en estructura de 32 bits que utilice exponentes en base 16:
   - $(675.125)_{10}$
   - $(- .00125)_{10}$
   - $(5008)_{10}$

3. Represente en estructura de 16 bits, en signo magnitud, signo complemento a uno y signo complemento a dos los siguientes enteros:
   - $(32675)_{10}$
   - $(-564)_{10}$
   - $(8213)_{10}$

3

EVALUACION DE ALGORITMOS

La evaluación de algoritmos consiste en medir la eficiencia de un algoritmo en cuanto a consumo de memoria y tiempo de ejecución.

Anteriormente, cuando existían grandes restricciones tecnológicas de memoria, evaluar un algoritmo en cuanto a consumo de recursos de memoria era bastante importante, ya que dependiendo de ella los esfuerzos de programación y de ejecución serían grandes. En la actualidad, con el gran desarrollo tecnológico del hardware, las restricciones de consumo de memoria han pasado a un segundo plano, por lo tanto, el aspecto de evaluar un algoritmo en cuanto a su consumo de memoria no es relevante.

En cuanto al tiempo de ejecución, a pesar de las altas velocidades de procesamiento que en la actualidad existen, sigue siendo un factor fundamental, especialmente en algoritmos que tienen que ver con el control de procesos en tiempo real y en aquellos cuya frecuencia de ejecución es alta.

Procedamos entonces a ver cómo determinar el tiempo de ejecución de un algoritmo. Básicamente existen dos formas de hacer esta medición: una que llamamos *a posteriori* y otra que llamamos *a priori*.

Consideremos la primera forma: El proceso de elaboración y ejecución de un algoritmo consta de los siguientes pasos:

- Elaboración del algoritmo.
- Codificación en algún lenguaje de programación.
- Compilación del programa.
- Ejecución del programa en una máquina determinada.

Cuando se ejecuta el programa, los sistemas operativos proporcionan la herramienta de informar cuánto tiempo consumió la ejecución del algoritmo.

Esta forma de medición tiene algunos inconvenientes: al codificar el algoritmo en algún lenguaje de programación estamos realmente midiendo la eficiencia del lenguaje de programación, ya que no es lo mismo un programa codificado en FORTRAN, que en PASCAL o que en C; al ejecutar un programa en una máquina determinada estamos introduciendo otro parámetro, el cual es la eficiencia de la máquina, ya que no es lo mismo ejecutar el programa en un XT, en un 386, en un 486, en un pentium, en un AS400, en un RISC, etc.

En otras palabras con esta forma de medición estamos es evaluando la calidad del compilador y de la máquina, mas no la del algoritmo, que es nuestro interés.

Lo que nos interesa es medir la eficiencia del algoritmo por el hecho de ser ese algoritmo, independiente del lenguaje en el que se haya codificado y de la máquina en la cual se ejecute.

Veamos entonces la forma de medir la ejecución de un algoritmo *a priori*.

Para ello necesitamos definir dos conceptos básicos que son: **contador de frecuencias** y **orden de magnitud**.

El **contador de frecuencias** es una expresión algebraica que indica el número de veces que se ejecutan las instrucciones de un algoritmo. Para ilustrar cómo determinar esta expresión consideremos los siguientes ejemplos:

ALGORITMO 1

1.	read(a,b,c)	1
2.	x = a + b	1
3.	y = a + c	1
4.	z = b * c	1
5.	w = x / y - z	1
6.	write(a,b,c,w)	1
	Contador de frecuencias =	6

ALGORITMO 2

1.	read(n)	1
2.	s = 0	1
3.	i = 1	1
4.	while i <= n do	n + 1

5.	s = s + 1	n
6.	i = i + 1	n
7.	end(while)	n
8.	write(n,s)	1
	Contador de frecuencias =	4n + 5

ALGORITMO 3

1.	read(n,m)	1
2.	s = 0	1
3.	i = 1	1
4.	while i <= n do	n + 1
5.	t = 0	n
6.	j = 1	n
7.	while j <= m do	n x m + n
8.	t = t + 1	n x m
9.	j = j + 1	n x m
10.	end(while)	n x m
11.	s = s + t	n
12.	i = i + 1	n
13.	end(while)	n
14.	write(n,m,s)	1
	Contador de frecuencias =	4n x m + 7n + 5

ALGORITMO 4

1.	read(n,m)	1
2.	s = 0	1
3.	i = 1	1
4.	while i <= n do	n + 1
5.	t = 0	n
6.	j = 1	n
7.	while j <= n do	$n^2 + n$
8.	t = t + 1	n^2
9.	j = j + 1	n^2
10.	end(while)	n^2
11.	s = s + t	n
12.	i = i + 1	n
13.	end(while)	n
14.	write(n,m,s)	1
	Contador de frecuencias =	$4n^2 + 7n + 5$

ALGORITMO 5

1.	read(n,m)	1
2.	s = 0	1
3.	i = 1	1
4.	while i <= n do	n + 1
5.	s = s + 1	n
6.	i = i + 1	n
7.	end(while)	n
8.	write(n,s)	1
9.	s = 0	1
10.	i = 1	1
11.	while i <= m do	m + 1
12.	t = t + 1	m
13.	i = i + 1	m
14.	end(while)	m
15.	write(m,s)	1
	Contador de frecuencias =	4n + 4m + 9

En todos los algoritmos las instrucciones están numeradas secuencialmente. Al frente de cada instrucción aparece un número o una letra que indica el número de veces que se ejecuta esa instrucción en el algoritmo.

En el **algoritmo 1** cada instrucción se ejecuta sólo una vez. El total de veces que se ejecutan todas las instrucciones es 6. Este valor es el contador de frecuencias para el algoritmo 1.

En el **algoritmo 2** las instrucciones 1, 2, 3 y 8 se ejecutan sólo una vez, mientras que las instrucciones 5, 6 y 7 se ejecutan **n** veces cada una, ya que pertenecen a un ciclo, cuya variable controladora del ciclo se inicia en uno, tiene un valor final de **n** y se incrementa de a uno, la instrucción 4 se ejecuta una vez más ya que cuando **i** sea mayor que **n** de todas formas hace la pregunta. Por tanto, el número de veces que se ejecutan las instrucciones del algoritmo es 4n + 5. Esta expresión es el contador de frecuencias para el algoritmo 2.

De una manera análoga se obtuvieron los contadores de frecuencias para los algoritmos 3, 4 y 5.

Veamos ahora cómo obtener los órdenes de magnitud para cada uno de los algoritmos dados.

El **orden de magnitud** es el concepto que define la eficiencia del algoritmo en cuanto a tiempo de ejecución. Se obtiene a partir del contador de frecuencias: se

eliminan los coeficientes, las constantes y los términos negativos, de los términos resultantes: si son dependientes entre sí, se elige el mayor de ellos y éste será el orden de magnitud de dicho algoritmo, de lo contrario el orden de magnitud será la suma de los términos que quedaron. Si el contador de frecuencias es una constante, como en el caso del algoritmo 1, el orden de magnitud es O(1).

Teniendo presente esto, los órdenes de magnitud de los algoritmos dados son:

Para el algoritmo 1: Orden de magnitud O(1).
Para el algoritmo 2: Orden de magnitud O(n).
Para el algoritmo 3: Orden de magnitud O(nxm).
Para el algoritmo 4: Orden de magnitud $O(n^2)$.
Para el algoritmo 5: Orden de magnitud O(n+m).

Es bueno aclarar lo siguiente: en los ejemplos, el valor de **n** es un dato que se lee dentro del programa. En el caso más amplio **n** podrá ser el número de registros que haya que procesar en un archivo, el número de datos que haya que producir como salida, o quizá otro parámetro diferente. Es decir, hay que poner atención en el análisis que se haga del algoritmo, de cuál es el parámetro que tomamos como **n**.

En el ambiente de sistemas los órdenes de magnitud más frecuentes para los algoritmos que se desarrollan son:

ORDEN DE MAGNITUD	REPRESENTACION
Constante	O(1)
Lineal	O(n)
Cuadrático	$O(n^2)$
Cúbico	$O(n^3)$
Logarítmico	$O(\log_2(n))$
Semilogarítmico	$O(n\log_2(n))$
Exponencial	$O(2^n)$

Si quisiéramos clasificar estos órdenes de magnitud en forma ascendente en cuanto a eficiencia tendríamos: los algoritmos más eficientes son aquellos con orden de magnitud constante, luego los de orden de magnitud logarítmico, a continuación los de orden de magnitud lineal, luego los semilogarítmicos, luego los cuadráticos, luego los cúbicos y por último los algoritmos con orden de magnitud exponencial.

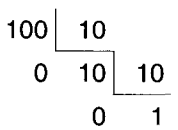
Hasta aquí hemos visto algoritmos cuyo orden de magnitud es constante, lineal y cuadrático. No es difícil imaginar un algoritmo cuyo orden de magnitud sea cúbico.

Pasemos a tratar algoritmos con orden de magnitud logarítmico. Empecemos definiendo lo que es un logaritmo.

La definición clásica del logaritmo de un número **x** es: es el exponente al cual hay que elevar un número llamado base para obtener **x**.

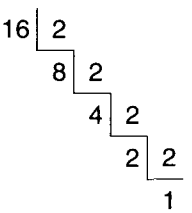
Con fines didácticos, presentamos otra definición de logaritmo: logaritmo de un número **x** es el número de veces que hay que dividir un número, por otro llamado base, para obtener como cociente uno (1).

Por ejemplo, si tenemos el número 100 y queremos hallar su logaritmo en base 10, habrá que dividirlo por 10 sucesivamente hasta obtener como cociente uno (1).



hubo que dividir el 100 dos veces por 10 para obtener cociente 1, por tanto el logaritmo en base 10 de 100 es 2.

Si queremos hallar el logaritmo en base 2 de 16 habrá que dividir 16 sucesivamente por 2 hasta obtener un cociente de 1. veamos:



es decir, hubo que dividir el 16, cuatro (4) veces por dos (2) para obtener un cociente de uno (1), por tanto el logaritmo en base 2 de 16 es 4.

Planteemos un ejemplo práctico de algoritmos en el cual se presente esta situación.

Consideremos el caso de un vector en el cual tenemos los datos ordenados ascendentemente:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	2	4	7	9	11	15	17	28	31	36	43	50	58	62	69

Si nos interesa buscar el dato **X = 85** y decir en cuál posición del vector se encuentra hay dos formas de hacerlo:

Una, es hacer una búsqueda secuencial a partir del primer elemento e ir comparando el contenido de esa posición con el dato **X (X = 85)**, el proceso terminará cuando lo encontremos o cuando hayamos recorrido todo el vector y no lo encontramos. Un algoritmo que haga ese proceso es el siguiente:

Sea **n** el número de elementos del vector y **x** el dato a buscar.

```
i = 1
while i <= n  and  v(i) <> x  do
        i = i + 1
end(while)
if  i < = n  then
        pos = i
else
          write('el dato ',x, 'no existe')
end(if)
```

En dicho algoritmo tenemos planteado un ciclo, el cual, en el peor de los casos se ejecuta **n** veces, es decir, cuando el dato **x** no esté en el vector. Por tanto el orden de magnitud de ese algoritmo es **O(n).**

En general, cuando se efectúa una búsqueda, el peor caso es no encontrar lo que se está buscando.

En nuestro ejemplo hubo que hacer 15 comparaciones para poder decir que el 85 no está en el vector.

Si el vector hubiera tenido un millón de datos y hubiéramos tenido que buscar un dato que no estaba, hubiéramos tenido que haber hecho un millón de comparaciones para poder decir que el dato buscado no se encontró.

Ahora, si aprovechamos la característica de que los datos en el vector están ordenados podemos plantear otro método de búsqueda:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	2	4	7	9	11	15	17	28	31	36	43	50	58	62	69
								M							

Comparamos el dato **x** con el dato del elemento de la mitad del vector, (llamémoslo **M**). Pueden suceder tres casos: **V(M) = x; V(M) > x; V(M) < x**

Si **V(M) = x** se ha terminado la búsqueda.

Si **V(M) > x** significa que si el dato **x** se encuentra en el vector **V**, estará en la primera mitad del vector.

Si **V(M) < x** significa que si el dato **x** se encuentra en el vector **V**, estará en la segunda mitad del vector.

En nuestro ejemplo, para seguir buscando el dato **x** = 85 lo haremos desde la posición 9 hasta la posición 15.

Es decir, con una comparación hemos eliminado la mitad de los datos sobre los cuales hay que efectuar la búsqueda. En otras palabras, hemos dividido la muestra por dos (2).

El conjunto de datos sobre el cual hay que efectuar la búsqueda queda así:

9	10	11	12	13	14	15
31	36	43	50	58	62	69
			M			

De la mitad que quedó escogemos nuevamente el elemento de la mitad y repetimos la comparación planteada anteriormente. Con esta comparación eliminamos nuevamente la mitad de los datos que nos quedaban, es decir, dividimos nuevamente por dos.

Si continuamos este proceso, con cuatro comparaciones podremos afirmar que el dato **x** = 85 no está en el vector.

Hemos reducido el número de comparaciones de 15 a 4.

Cuatro (4) es el logaritmo en base 2 de 15, aproximándolo por encima.

Si **N** fuera 65000, con 16 comparaciones podremos decir que un dato no se encuentra en un conjunto de 65000 datos.

Como puede observarse, la ganancia en cuanto al número de comparaciones es grande.

Un algoritmo de búsqueda que funcione con esta técnica tiene orden de magnitud logarítmico en base dos, ya que la muestra de datos se divide sucesivamente por dos.

Consideremos ahora, algoritmos sencillos cuyo orden de magnitud es logarítmico:

```
n = 32
s = 0
i = 32
while i > 1 do
        s = s + 1
        i = i / 2
end(while)
write(n,s)
```

En el algoritmo anterior, la variable controladora del ciclo es **i**, y dentro del ciclo, **i** se divide por dos (2). Si hacemos un seguimiento detallado tendremos:

La primera vez que se ejecuta el ciclo la variable **i** sale valiendo 16.
La segunda vez que se ejecuta el ciclo la variable **i** sale valiendo 8.
La tercera vez que se ejecuta el ciclo la variable **i** sale valiendo 4.
La cuarta vez que se ejecuta el ciclo la variable **i** sale valiendo 2.
La quinta vez que se ejecuta el ciclo la variable **i** sale valiendo 1.
y termina el ciclo. Es decir, las instrucciones del ciclo se ejecutan 5 veces.

Cinco (5) es el logaritmo en base dos (2) de **n** (**n** = 32), por consiguiente, cada instrucción del ciclo se ejecuta un número de veces igual al logaritmo en base dos de **n**. El contador de frecuencias y el orden de magnitud de dicho algoritmo se presentan a continuación.

n = 32	1
s = 0	1
i = 32	1
while i > 1 do	$\log_2 n + 1$
s = s + 1	$\log_2 n$
i = i / 2	$\log_2 n$
end(while)	$\log_2 n$
write(n,s)	1
Contador de frecuencias =	$4\log_2 n + 5$

Eliminando constantes y coeficientes el orden de magnitud es $O(\log_2 n)$
Consideremos este nuevo algoritmo fuera:

```
n = 32
s = 0
i = 1
while i < n do
        s = s + 1
        i = i * 2
end(while)
write(n,s)
```

En este algoritmo la variable controladora del ciclo **i** se multiplica por dos dentro del ciclo.

Haciendo un seguimiento tendremos:

En la primera pasada la variable **i** sale valiendo 2.
En la segunda pasada la variable **i** sale valiendo 4.
En la tercera pasada la variable **i** sale valiendo 8.
En la cuarta pasada la variable **i** sale valiendo 16.
En la quinta pasada la variable **i** sale valiendo 32.
y el ciclo se termina.

Las instrucciones del ciclo se ejecutan 5 veces, por tanto el orden de magnitud del algoritmo es logarítmico en base dos (2).

En general, para determinar si un ciclo tiene orden de magnitud lineal o logarítmico, debemos fijarnos cómo se está modificando la variable controladora del ciclo: si ésta se modifica mediante sumas o restas el orden de magnitud del ciclo es lineal, si se modifica con multiplicaciones o divisiones el orden de magnitud del ciclo es logarítmico.

Consideremos ahora el siguiente algoritmo:

```
n = 81
s = 0
i = 81
while  i > 1  do
        s = s + 1
        i = i / 3
end(while)
write(n,s)
```

Aquí, la variable controladora del ciclo **i** se divide por tres dentro del ciclo.

Haciéndole seguimiento tendremos:

En la primera pasada la variable **i** sale valiendo 27.
En la segunda pasada la variable **i** sale valiendo 9.
En la tercera pasada la variable **i** sale valiendo 3.
En la cuarta pasada la variable **i** sale valiendo 1.
y termina el ciclo.

Las instrucciones del ciclo se ejecutaron 4 veces. Cuatro es el logaritmo en base 3 de 81. Por consiguiente el orden de magnitud de dicho algoritmo es logarítmico en base 3.

En general, si tenemos un algoritmo:

```
n = 81
s = 0
i = 81
while i > 1 do
        s = s + 1
        i = i / x
end(while)
write(n,s)
```

Aquí, la variable controladora del ciclo se divide por **x**.

El número de veces que se ejecutan las instrucciones del ciclo es logaritmo en base **x** de **n**, por tanto el orden de magnitud es logarítmico en base **x**.

Para obtener algoritmos con orden de magnitud semilogarítmico basta con que un ciclo logarítmico se ubique dentro de un ciclo con orden de magnitud O(n) o viceversa.

Por ejemplo:

read(n)	1
s = 0	1
i = 1	1
while i <= n do	n + 1
t = 0	n
j = n	n
while j > 1 do	$n\log_2 n + n$
t = t + 1	$n\log_2 n$
j = j / 2	$n\log_2 n$
end(while)	$n\log_2 n$
write(t)	n
s = s + t	n
i = i + 1	n
end(while)	n
write(n,s)	1
Contador de frecuencias =	$8n + 4n\log_2 n + 5$

y el orden de magnitud es semilogarítmico.

Digamos que la idea es elaborar algoritmos lo más eficiente posible. Consideremos el siguiente problema:

Elaborar algoritmo que lea un entero positivo **n** y que determine la suma de los enteros desde uno hasta **n**.

Una solución es la siguiente:

```
read(n)
s = 0
i = 1
while  i <= n  do
       s = s + i
       i = i + 1
end(while)
write(n,s)
```

El anterior algoritmo lee un dato entero **n** (**n > 0**), calcula e imprime la suma de los enteros desde 1 hasta **n**, y el orden de magnitud de él es **O(n).**

Matemáticamente hablando la suma de los enteros desde 1 hasta **n** es:

$$S = \sum_{i=1}^{i=n} i$$

sumatoria, que según los matemáticos es: $\frac{n*(n+1)}{2}$

Conociendo esta fórmula, podemos elaborar otro algoritmo que ejecute exactamente la misma tarea. Veamos:

```
read(n)
s = n*(n+1)/2
write(n,s)
```

el cual, tiene orden de magnitud O(1).

Es decir, tenemos dos algoritmos completamente diferentes que ejecutan exactamente la misma tarea, uno con orden de magnitud **O(n)** y el otro con orden de magnitud **O(1).**

Lo anterior no significa que toda tarea podrá ser escrita con algoritmos O(1), no, sino que de acuerdo a los conocimientos que se tengan o a la creatividad de cada cual, se podrán elaborar algoritmos más eficientes.

Este ejemplo, el cual es válido, desde el punto de vista de desarrollo de algoritmos, es aplicable también a situaciones de la vida real. Seguro que alguna vez usted tuvo algún problema, y encontró una solución, y se salió del problema, aplicando la solución que encontró, y sin embargo, al tiempo, dos o tres meses después, pensando en lo que había hecho, encontró que pudo haber tomado una mejor determinación y que le habría ido mejor.

La enseñanza es que cuando se nos presente un problema debemos encontrar como mínimo dos soluciones y luego evaluar cuál es la más apropiada.

Ya tenemos herramientas para determinar la solución más eficiente.

Como un ejemplo adicional consideremos el siguiente algoritmo:

```
1     read(n)
2     s = 0
3     for  i = 1 to n  do
4               t = 0
5               for  j = 1 to i  do
6                         t = t + 1
7               end(for)
8               s = s + t
9     end(for)
10    write(n,s)
```

Vamos a determinar el contador de frecuencias y el orden de magnitud de dicho algoritmo.

Las instrucciones 1, 2 y 10 se ejecutan sólo una vez.
Las instrucciones 3, 4, 8 y 9 se ejecutan **n** veces.

Consideremos ahora las instrucciones 5, 6 y 7.:

Cuando la **i** vale 1 las instrucciones 5, 6 y 7 se ejecutan 1 vez.
Cuando la **i** vale 2 las instrucciones 5, 6 y 7 se ejecutan 2 veces.
Cuando la **i** vale 3 las instrucciones 5, 6 y 7 se ejecutan 3 veces.
Cuando la **i** vale 4 las instrucciones 5, 6 y 7 se ejecutan 4 veces.
" " " " " " " " "
" " " " " " " " "

" " " " " " " " " "

Cuando la **i** vale **n**, las instrucciones 5, 6 y 7 se ejecutan **n** veces.

Es decir, el total de veces que se ejecutan las instrucciones 5, 6 y 7 es:

$$1 + 2 + 3 + ... + n$$

o sea la sumatoria de **i**, con **i** desde 1 hasta **n**, que como ya habíamos visto es n*(n+1)/2. Por tanto el contador de frecuencias de nuestro algoritmo será:

1	read(n)	1
2	s = 0	1
3	for i = 1 to n do	n + 1
4	t = 0	n
5	for j = 1 to i do	n(n+1)/2 + n
6	t = t + 1	n(n+1)/2
7	end(for)	n(n+1)/2
8	s = s + t	n
9	end(for)	n
10	write(n,s)	1
	contador de frecuencias =	3n(n+1)/2 + 5n + 4
	y el orden de magnitud es $O(n^2)$	

EJERCICIOS PROPUESTOS

1. Determine contador de frecuencias y orden de magnitud para los siguientes algoritmos:

1.1
```
sub_programa  p1(n)
        s = 0
        for  i = 1 to n  do
                for  j = 1 to i  do
                        for  k = 1 to j  do
                                s = s + 1
                        end(for)
                end(for)
        end(for)
fin(sub_programa)
```

```
1.2  sub_programa  p2(n)
          s = 0
          i = 1
          j = 1
          while  i <= n  and  j <= n  do
                s = s + 1
                i = i + 1
                if  i > n  and  j < n  then
                      i = 1
                      j = j + 1
                end(if)
          end(while)
          write(s)
     fin(sub_programa)


1.3  sub_programa  p3(vec,n,x)
          i = 1
          j = n
          do
                k = (i+j)/2
                if  vec(k) <= x  then
                      i = k + 1
                else
                      j = k - 1
                end(if)
          while  i <= j
     fin(sub_programa).


1.4  sub_programa  p4(vec,n)
          i = 1
          while  i <= n  do
                s = 0
                j = i + 1
                while  j <= n  and  j <= i + vec(i)  do
                      s = s + vec(i)
                      j = j + 1
                end(while)
                write(s)
                i = j
          end(while)
     fin(sub_programa)
```

```
1.5   Sub_programa p5()
          read(n)
          s = 0
          i = 2
          while i <= n do
                s = s + 1
                i = i * i
          end(while)
          write(n,s)
      fin(sub_programa)


1.6   Sub_programa p5()
          read(n)
        ' s = 0
          i = 3
          while i <= n do
                s = s + 1
                i = i * i
          end(while)
          write(n,s)
      fin(sub_programa)
```

4

MANEJO DINÁMICO DE MEMORIA

4.1 INTRODUCCIÓN

Para entrar a hablar del manejo dinámico de memoria consideremos brevemente lo que es el manejo estático de la memoria, es decir, los arreglos. Tratemos el caso de un vector, el cual llamaremos **V**.

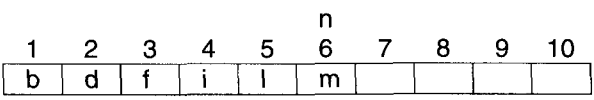


Figura 4.1

En el vector de la figura 4.1 manejamos un conjunto de datos ordenados ascendentemente.

Las operaciones básicas que se realizan sobre ese conjunto de datos son recorrer, insertar un dato y borrar un dato. Analicemos cada una de ellas.

Para recorrer el vector utilizamos una variable auxiliar, la cual generalmente la llamamos **i** y cuyo valor inicial es 1. Imprimimos el dato de la posición **i** (**write V(i)**) y avanzamos en el vector (**i = i+1**)

Si se desea insertar el dato **'g'** en el vector de la figura 4.1 el proceso a seguir es:

1. Buscar en cuál posición insertarlo.
2. Insertarlo en la posición correspondiente.

Del primer paso se obtiene que la posición en la cual hay que insertarlo es la posición 4 del vector.

El segundo paso deberá mover los datos desde la posición 4 hasta la posición 6, una posición hacia la derecha y luego asignar a la posición 4 el dato **'g'**.

Si el dato a insertar hubiera sido la letra **'a'** hubiéramos tenido que mover todos los datos del vector.

Si generalizamos, y decimos que el vector tiene **n** datos, en el peor de los casos, hay que mover **n** datos en el vector, o sea, cuando el dato a insertar deba quedar de primero, lo cual implica que el algoritmo de inserción tendrá orden de magnitud **O(n).**

Si deseamos borrar un dato de un vector los pasos a seguir son:

1. Buscar en cuál posición se halla el dato a borrar.
2. Borrar el dato del vector.

En caso de que el dato a borrar fuera la letra **'f'**, el primer paso me retorna 3, o sea, la posición en la cual se halla la letra **'f'**.

El segundo paso moverá los datos desde la posición 4 hasta la 6 una posición hacia la izquierda.

Si el dato a borrar hubiera estado en la posición 1 del vector y el vector tiene **n** datos, entonces habrá que mover **n-1** datos hacia la izquierda, y el orden de magnitud de dicho algoritmo es **O(n).**

De lo expuesto anteriormente, los algoritmos de inserción y borrado tienen orden de magnitud **O(n),** y esto, en el manejo de grandes volúmenes de información, con operaciones de alta frecuencia, como son insertar y borrar se considera ineficiente.

Por lo tanto se ha buscado una forma alterna de representación en la cual las operaciones de inserción y borrado sean eficientes, es decir, tengan orden de magnitud **O(1).**

4.2 CONCEPTO DE LISTA LIGADA

Consideremos los siguientes dos vectores que llamamos DATO Y LIGA.

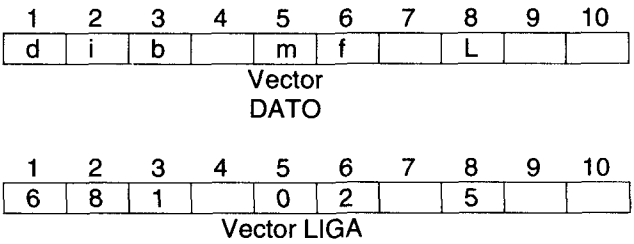


Figura 4.2

En el vector DATO almacenamos los datos en posiciones aleatorias.

Físicamente los datos se hallan en desorden. Nos interesa tenerlos ordenados lógicamente.

Para ello debemos conocer en cuál posición del vector se halla el primer dato. En nuestro ejemplo, el primer dato se halla en la posición 3 del vector. Utilizaremos una variable, llamémosla **L**, cuyo valor es 3. Es decir, el hecho de que **L** valga 3 significa: el primer dato se halla en la posición 3 del vector DATO.

Nos interesa saber en cuál posición se halla el siguiente dato, para ello utilizamos la posición 3 del vector LIGA.

El siguiente dato, que es la 'd', se halla en la posición 1 del vector DATO, por tanto en la posición 3 del vector LIGA tendremos un 1.

El hecho de que en la posición 3 del vector LIGA haya un 1 significa que el siguiente dato se halla en la posición 1 del vector DATO.

Para conocer el siguiente a la 'd' utilizamos la posición 1 del vector LIGA. Allí encontramos un 6, lo que significa que el siguiente dato a la 'd' se encuentra en la posición 6 del vector DATO.

En general, para un dato que se halle en la posición **i** del vector DATO, la correspondiente posición **i** del vector LIGA indica en cuál posición del vector DATO se halla el siguiente dato.

Los textos de computadores acostumbran presentar la situación descrita anteriormente, de la siguiente forma:

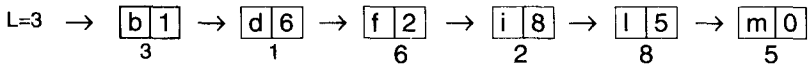


Figura 4.3

En general, cada cuadrito

DATO	LIGA

representa un registro o nodo, el cual consta de dos campos: uno que llamamos DATO y otro que llamamos LIGA. Si tenemos un registro **x**:

3.1416	7

x = 19

y nos queremos referir a sus campos lo haremos así: **dato(x)** y **liga(x)**

En el ejemplo, cuando hagamos mención a **dato(x)**, nos estamos refiriendo a 3.1416 y si hacemos mención a **liga(x)** nos estamos refiriendo a 7.

4.3 OPERACIONES SOBRE LISTAS LIGADAS

Consideremos ahora las operaciones fundamentales sobre listas ligadas:

- Recorrer la lista ligada.
- Insertar un registro en la lista ligada.
- Borrar un registro de la lista ligada.

Recorrer la lista ligada

Consideremos la lista de la figura 4.3, y digamos que sólo nos interesa imprimir los datos que hay en ella.

Para lograr esto requerimos de una variable auxiliar, llamémosla **p**.

Inicialmente, a **p** le asignamos el valor de **L**. **p** queda valiendo 3.

Para imprimir el contenido del campo de **dato** del registro **p** escribimos **write(dato(p))** y escribe el dato 'b'.

Para avanzar sobre la lista, es decir, trasladarnos al siguiente registro escribimos: **p = liga(p)**, es decir, a la variable **p** le asigna lo que hay en el campo de liga del registro **p**. **p** queda valiendo 1.

Imprimimos el dato del registro **p** y continuamos repitiendo el proceso hasta que **p** sea cero.

Un algoritmo que ejecuta esta tarea es:

```
Sub_programa recorre_lista(L)
        p = L
        while  p <> 0  do
                write(dato(p))
                p = liga(p)
        end(while)
fin(recorre_lista)
```

Insertar un registro en la lista ligada

Para insertar un registro en una lista ligada se requiere conocer a continuación de cuál registro es que hay que efectuar la inserción. Llamemos **y** este registro, y llamemos **x** el registro a insertar. Consideremos la figura 4.4.

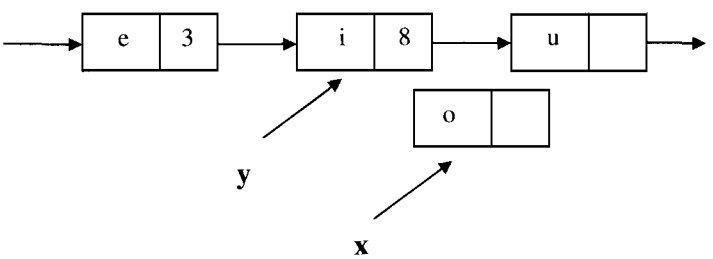
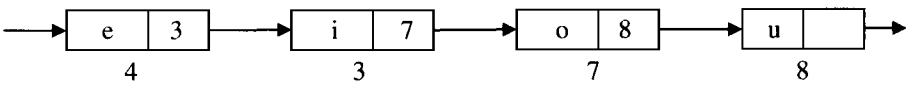


Figura 4.4

Insertar el registro **x** (**x = 7**), a continuación del registro **y** (**y = 3**) implica que cuando lleguemos al registro **y** debemos trasladarnos hacia el registro **x**, o sea que el campo de liga del registro **y** debe quedar valiendo 7, y cuando estemos en el registro **x** debemos trasladarnos hacia el registro 8, es decir, que el campo de liga del registro **x** debe quedar valiendo 8. Para lograr lo anterior debemos ejecutar las siguientes instrucciones:

liga(x) = liga(y)
liga(y) = x

y la lista queda:



Estrictamente hablando, esas son las únicas instrucciones necesarias para insertar un registro **x** a continuación de un registro **y** en una lista ligada.

Planteemos ahora el problema más completo.

Tenemos una lista ligada, se leyó un dato y hay que insertarlo en la lista.

Los pasos a seguir son:

- Buscar dónde insertar el dato leído, es decir, determinar **y**.
- Conseguir un registro **x**, guardar el dato leído en **x** y luego insertarlo a continuación del registro **y**.

Ocupémonos del primer paso. Consideremos la siguiente lista:

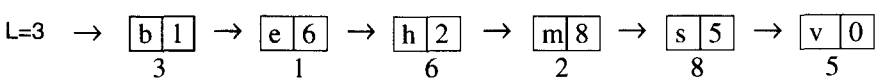


Figura 4.5

Sea **d** = 'j' el dato leído.

Para buscar dónde insertarlo debemos recorrer la lista comparando el dato de cada registro con el dato leído hasta encontrar un registro cuyo dato sea mayor que **d**, el dato leído.

El nuevo registro debe insertarse antes de ese registro, o sea, a continuación del registro anterior a aquel que tiene un dato mayor. Esto significa que si utilizamos una variable **p** para hacer el recorrido e ir efectuando las comparaciones, debemos utilizar otra variable que permanentemente apunte hacia el registro anterior a **p**. Llamemos esta variable **y**.

Inicialmente a **p** le asignamos el valor de **L**, y como éste es el primer registro y no tiene anterior, el valor inicial de **y** es cero.

Para efectuar esta tarea utilizaremos la siguiente función:

```
function buscar_donde_insertar(L, d)
        p = L
        y = 0
        while  p <> 0  and dato(p) < d  do
                y = p
                p = liga(p)
        end(while)
return(y)
fin(buscar_donde_insertar)
```

L y **d** son parámetros de entrada, **y** es el dato de retorno.

Es bueno hacer notar en este algoritmo que si el dato a insertar debe quedar de primero en la lista, el valor retornado en **y** es cero; si el dato a insertar debe quedar de último en la lista, entonces **y** quedará apuntando hacia el último registro de la lista ligada.

A continuación presentamos el algoritmo de inserción. Observe que dicho algoritmo no requiere mover datos para poder insertar un dato en la posición apropiada. Esto hace que el orden de magnitud de dicho algoritmo sea constante, es decir, **O(1).**

```
sub_programa insertar(L, y, d)
        conseguir_registro(x)
        dato(x) = d
        if  y = 0  then
                liga(x) = L
                L = x
        else
                liga(x) = liga(y)
                liga(y) = x
        end(if)
fin(insertar)
```

y y **d** son parámetros de entrada, **L** es parámetro de entrada y salida.

En este algoritmo utilizamos una instrucción **conseguir_registro(x)**, la cual, invoca un sub_programa que interactúa con el sistema operativo solicitándole un registro, cuya dirección la retorna en la variable **x**. Más adelante hablaremos en detalle acerca de este sub_programa.

Borrar un registro de una lista ligada

El concepto fundamental de borrar un registro de una lista ligada es: borrar un registro **x** de una lista ligada consiste en desconectarlo de la lista. Si tenemos la lista de la figura 4.6:

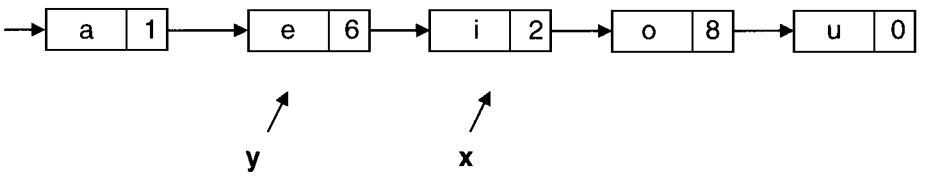


Figura 4.6

y queremos borrar el registro **x** (x = 2), significa que cuando estemos ubicados en el registro 1 **(y = 1),** debemos pasarnos hacia el registro 2, en vez de hacia el registro 6. Esto implica que debemos conocer cuál es el registro anterior a **x**. Llamemos entonces **y**, el registro anterior a **x**. La instrucción para desconectar el registro **x** de la lista ligada es simplemente:

liga(**y**) = liga(**x**)

Estrictamente hablando, esa es la única instrucción necesaria para borrar un registro **x** de una lista ligada. Consideremos el proceso completo.

Sea **d** la variable que contiene un dato leído que se desea borrar de una lista ligada **L**. Los pasos a seguir son:

- Determinar **x** e **y**.
- Borrar **x**.

Para determinar **x** e **y** basta recorrer la lista comparando el dato del registro **x** con el dato leído **d** hasta que sean iguales. Cuando esto suceda hemos determinado los valores de **x** e **y**.

Inicialmente el valor que se le asigna a **x** es **L** y el valor de **y** es cero.

El algoritmo es:

```
sub_programa buscar_dato_a_borrar(L, d, x, y)
        x = L
        y = 0
        while  x <> 0  and  dato(x) <> d  do
                y = x
                x = liga(x)
        end(while)
fin(buscar_dato_a_borrar)
```

En el anterior algoritmo los parámetros **L** y **d** son parámetros de entrada, mientras que **x** e **y** son parámetros de salida, los cuales significan: **x** registro a borrar e **y** registro anterior a **x**. Nuestro algoritmo retornará **y = 0** cuando el registro a borrar sea el primero de la lista ligada, además retornará **x = 0** cuando el dato a borrar no exista.

El algoritmo de borrado es:

```
sub_programa borrar(L, x, y)
        if  x = 0  then
            write('dato a borrar no existe')
            return
        end(if)
        if  y = 0  then
            L = liga(L)
        else
            liga(y) = liga(x)
        end(if)
        liberar_registro(x)
fin(borrar)
```

En este algoritmo presentamos la instrucción **liberar_registro(x)**, la cual es la llamada a un sub_programa que interactúa con el sistema operativo indicándole que puede disponer del registro **x**, ya que nuestro programa no lo necesita. Si no hacemos esto, el sistema operativo sigue creyendo que ese registro está ocupado.

Como un ejemplo de utilización de estos algoritmos consideremos un subprograma que borre todos los datos que tengan un dato **d** en una lista simplemente ligada **L**.

```
sub_programa  borra_todos(L, d)
        buscar_dato_a_borrar(L, d, x, y)
        while  x <> 0  do
                borrar(L, x, y)
```

```
            buscar_dato_a_borrar(L, d)
        end(while)
    fin(borra_todos)
```

4.4. CONSTRUCCIÓN DE LISTAS LIGADAS

Pasemos ahora a considerar cómo construir listas ligadas.

Básicamente hay tres formas de construir una lista ligada. una, en que los datos queden ordenados a medida que la lista se va construyendo; dos, insertando registros siempre al final de la lista y tres, insertando los registros siempre al principio de la lista.

Para la primera forma de construcción nuestro algoritmo es:

```
L= 0
mientras haya datos por leer haga
        lea(d)
        y = buscar_donde_insertar(L,d)
        insertar(L, y, d)
fin(mientras)
```

Es decir, basta con plantear un ciclo para lectura de datos e invocar los sub_programa desarrollados anteriormente para buscar en cuál posición insertar un nuevo registro y luego insertarlo.

Para la segunda forma de construcción nuestro algoritmo es:

```
L = 0
mientras haya datos por leer haga
        lea(d)
        y = ultimo(L)
        insertar(L, y, d)
fin(mientras)
```

En el ciclo de dicho algoritmo invocamos una función **último(L)**, la cual recorre la lista para determinar el último registro de ella. El último registro de una lista ligada es aquel cuyo campo de liga es cero. Un algoritmo para hacer esto es:

```
function ultimo(L)
        if L = 0 then
                return(0)
        end(if)
```

```
        ultimo = L
        while liga(ultimo) <> 0 do
              ultimo = liga(ultimo)
        end(while)
        return(ultimo)
  fin(ultimo)
```

Si consideramos que la lista tiene **n** registros, el orden de magnitud de dicho algoritmo es **O(n)**. En otras palabras, por cada registro que haya que insertar hay que recorrer toda la lista para determinar cuál es el último registro. Lo anterior, además de impráctico es ineficiente, ya que sabemos que todo registro debe insertarse siempre al final de la lista.

Para obviar este problema utilizaremos una variable auxiliar, que llamaremos **ultimo**, que siempre apunte hacia el último registro de la lista ligada.

Utilizando esta variable **ultimo** las instrucciones para insertar un registro son:

```
conseguir_registro(x)
dato(x) = d
liga(x) = 0
liga(ultimo) = x
ultimo = x
```

y el algoritmo completo para construir la lista es:

```
lea(d)
conseguir_registro(x)
dato(x) = d
liga(x) = 0
L = x
ultimo = x
mientras haya datos por leer haga
      lea(d)
      conseguir_registro(x)
      dato(x) = d
      liga(x) = 0
      liga(ultimo) = x
      ultimo = x
fin(mientras)
```

Para la tercera forma de construcción nuestro algoritmo es:

```
L = 0
mientras haya datos por leer haga
   lead(d)
   insertar(L, 0, d)
fin(mientras)
```

Es decir, simplemente leemos cada dato e invocamos nuestro subprograma de insertar enviando en el parámetro **y** el valor de cero, lo cual indica que el dato a insertar debe quedar de primero en la lista.

Hasta aquí hemos desarrollado lo que es básicamente la herramienta lista ligada, con sus operaciones fundamentales: construir la lista, recorrer la lista, insertar un registro y borrar un registro.

4.5 INTERACCIÓN CON EL SISTEMA OPERATIVO

Veamos ahora un poquito referente a los sub_programas **conseguir_registro(x)** y **liberar_registro(x)**. Parte de las funciones del sistema operativo de un computador es administrar la memoria de éste. Dicha administración incluye conocer permanentemente cuáles registros de memoria están libres y cuáles registros están ocupados, para que así, cuando algún programa que trabaje memoria en forma dinámica invoque alguno de estos procedimientos su interacción con el sistema operativo sea correcta y eficiente.

Dicha interacción consiste en que el sistema operativo le asignará memoria al programa cuando éste lo solicite y recibirá memoria cuando el programa la libere.

Conseguir_registro(x): El sistema operativo asigna memoria al programa que hizo la solicitud. La dirección del registro o bloque asignado retorna en el parámetro **x**. Para poder efectuar una asignación correcta, el sistema operativo debe conocer cuáles registros de memoria están libres y asignar uno de ellos.

Liberar_registro(x): Da la orden al sistema operativo de que disponga del registro o bloque **x** y lo marque como libre.

El sistema operativo maneja una lista ligada con los registros disponibles.

Si un programa invoca **conseguir_registro(x)** entonces el sistema operativo asignará el primer registro de la lista ligada en la cual maneja los registros libres, al programa que hizo la solicitud. Esta lista ligada consta de todos los registros

disponibles, y una vez asignado al programa, lo borrará de la lista de disponibles, es decir, lo desconecta de dicha lista.

Si se invoca **liberar_registro(x)**, el sistema operativo inserta un registro al principio de la lista de registros disponibles.

Las operaciones de inserción y borrado las hace en un solo extremo, lo cual significa que la lista de disponibles la trabaja como una pila, estructura que veremos más adelante. Los algoritmos para conseguir un registro y liberar un registro se presentan a continuación.

```
sub_programa conseguir_registro(x)
      if disp = 0 then
            write("no hay registros disponibles")
            stop
      end(if)
      x = disp
      disp = liga(disp)
fin(conseguir_registro)

sub_programa liberar_registro(x)
      liga(x) = disp
      disp = x
fin(liberar_registro)
```

De aquí en adelante, por simplicidad, el subprograma **conseguir_registro(x)** lo seguiremos invocando **new(x)** y el subprograma **liberar_registro(x)** lo invocaremos **free(x)**.

4.6 INTERCALACIÓN DE DOS LISTAS LIGADAS ORDENADAS

Consideremos ahora un algoritmo típico en el manejo de listas ligadas.

Sean **A** y **B** los apuntadores a dos listas ligadas, cada una de las cuales tiene datos únicos ordenados ascendentemente. El objetivo es construir una nueva lista ligada intercalando los datos de las listas **A** y **B** sin que queden datos repetidos en la lista resultado.

Para efectuar dicha tarea debemos recorrer las listas **A** y **B** simultáneamente, comparando los datos de ellas. Cuando el dato de un registro de la lista **A** sea menor que el dato de un registro de la lista **B**, añadimos un registro en la lista resultado con los datos correspondientes al registro de la lista **A** y avanzamos

sobre la lista **A**; si el dato menor corresponde al registro de la lista **B** entonces añadimos un registro a la lista resultado con los datos correspondientes al registro de la lista **B** y avanzamos sobre la lista **B**; en caso de que los datos sean iguales añadimos un registro a la lista resultado con ese dato y avanzamos sobre ambas listas.

Para recorrer una lista ligada se requiere de una variable auxiliar. Como aquí se requiere recorrer dos listas simultáneamente necesitaremos dos variables auxiliares. Llamémoslas **p** y **q**.

El sub_programa que efectúa esta tarea se presenta a continuación.

El algoritmo tiene tres parámetros. Los dos primeros son parámetros por valor y son los apuntadores a las listas que se desean intercalar, el tercer parámetro es por referencia y en él se retornará el apuntador hacia el primer registro de la lista resultado.

```
sub_programa intercala(A, B, C)
1           new(x)
2           L = x
3           ultimo = x
4           p = A
5           q = B
6           while  p <> 0  and  q <> 0  do
7                 casos
8                       :dato(p) < dato(q):
9                             añadir_registro(dato(p), ultimo)
10                            p = liga(p)
11                      :dato(p) > dato(q):
12                            añadir_registro(dato(q), ultimo)
13                            q = liga(q)
14                      :dato(p) = dato(q):
15                            añadir_registro(dato(p), ultimo)
16                            p = liga(p)
17                            q = liga(q)
18                fin(casos)
19          end(while)
20          while  p <> 0  do
21                añadir_registro(dato(p), ultimo)
22                p = liga(p)
23          end(while)
24          while  q <> 0  do
25                añadir_registro(dato(q),ultimo)
```

```
26                  q = liga(q)
27            end(while)
28            liga(ultimo) = 0
29            x = L
30            L = liga(L)
31            free(x)
fin(intercala)

sub_programa añadir_registro(d, u)
      new(x)
      dato(x) = d
      liga(u) = x
      u = x
fin(añadir_registro)
```

En dicho sub_programa utilizamos una variable **ultimo** ya que la forma de construir la lista resultado es añadiendo registros siempre al final de la lista. Utilizamos además un sub_programa llamado **añadir_registro**, el cual tiene como parámetros un campo de dato y el campo llamado **ultimo**. El campo de dato es parámetro por valor y el campo **ultimo** es parámetro por referencia.

Como una lista ligada se puede terminar de recorrer mientras que en la otra lista aún faltan registros, los ciclos de las instrucciones 20 y 24 controlan esas situaciones. Al final del algoritmo, instrucciones 28 y 30, borramos el primer registro de la lista ligada resultado, ya que este es un registro auxiliar que se utilizó para construir la lista más fácil y eficientemente. Cuando expliquemos el sub_programa **añadir_registro** trataremos este caso.

El sub_programa **añadir_registro** consigue un registro, le asigna el dato recibido, lo conecta con el último y dice que ese es el nuevo último.

Ahora, para qué utilizamos un registro auxiliar en la construcción de la lista resultado.

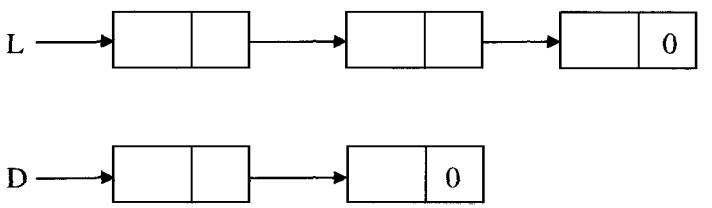
Si no utilizamos registro auxiliar, las tres primeras instrucciones del sub_programa **intercala** habría que reemplazarlas por una sola instrucción que fuera **ultimo = 0**, y la primera vez que se invoque el sub_programa **añadir_registro** el parámetro **u** entraría valiendo 0, lo cual implica que el sub_programa **añadir_termino** debe controlar esta situación. El sub_programa **añadir_termino** quedaría como sigue:

```
sub_programa añadir_registro(L, d, u)
        new(x)
        dato(x) = d
        if  u = 0  then
                L = x
        else
                liga(u) = x
        end(if)
        u = x
fin(añadir_registro)
```

y la pregunta **if u = 0** sólo sería verdadera la primera vez que se llame el sub_programa, todas las demás veces el resultado de la pregunta es falso. Por tanto, hacer esta pregunta es ineficiente, además que hay que codificar más instrucciones. Como se puede observar, con la utilización de la variable auxiliar, al principio de la lista obviamos la pregunta en **añadir_registro**.

4.7 LIBERACIÓN DE LISTAS LIGADAS

Sea **L** el apuntador hacia el primer registro de la lista que se desea liberar; sea **disp** el apuntador hacia el primer registro de la lista ligada de registros disponibles.



El siguiente algoritmo retorna todos los registros de la lista **L** a la lista de disponibles, uno por uno.

```
sub_programa liberar_lista(L)
     while L <> 0 do
          x = L
          L = liga(L)
          free(x)
     end(while)
fin(liberar_lista)
```

Así, que si la lista tiene **n** registros el algoritmo será de orden **O(n)**. Tener un algoritmo que devuelva, uno por uno, los registros de una lista ligada es ineficiente. Sería mucho más eficiente liberar todos los registros de una sola vez.

Si dispongo de la lista **L** y quiero llevar sus registros a la lista de disponibles **(disp)**, sin tener que hacerlo uno por uno, debe recorrerse la lista **L** para encontrar su último registro y ponerlo a apuntar hacia el primer registro de la lista **disp**. Luego la nueva lista **disp** será **L** y tendrá la misma dirección de memoria que **L**.

Veamos el algoritmo:

```
sub_programa liberar_lista(L)
      p = L
      while liga(p) <> 0 do
            p = liga(p)
      end(while)
      liga(p) = disp
      disp = L
       L = 0
fin(liberar_lista)
```

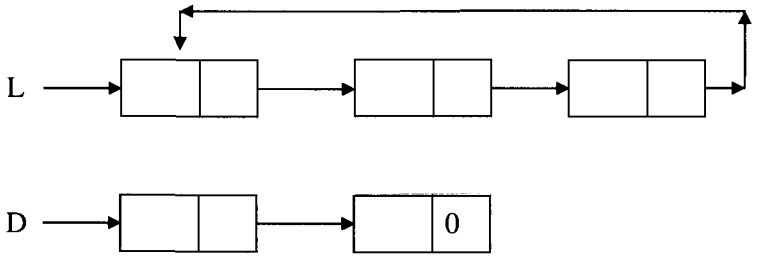
Este algoritmo devuelve todos los registros de una lista **L** a una lista de disponibles. El orden de magnitud es **O(n)** ya que de todas formas tenemos que recorrer la lista ligada para identificar el último registro.

Aunque ambos algoritmos tienen el mismo orden de magnitud, es más eficiente el segundo ya que dentro del ciclo sólo hay una instrucción, y se omite el llamado al sub_programa **liberar_registro(x)**.

Sin embargo, para procesos que se presentan con mucha frecuencia, lo ideal es tener un algoritmo con orden de magnitud **O(1)**. Para obtener esto, basta con hacer una pequeña modificación a la representación en lista ligada: hacerla circular, o sea que el campo de liga del último registro ya no será cero sino que apuntará hacia el primer registro de la lista ligada.

El sub_programa de liberar una lista en representación de lista ligada circular tendrá orden de magnitud **O(1)**.

El proceso consiste en encadenar **disp** (lista de disponibles) al primer registro de la lista **L** y asignarle a **disp** el valor del segundo registro de la lista **L**.



Para lograr dicho objetivo se guarda en una variable **x** el segundo registro de la lista **L**. La variable **disp** apuntará hacia este registro y el campo liga del registro **L** apuntará hacia el registro que inicialmente era **disp**.

```
sub_programa liberar_lista_circular(L)
        x = liga(L)
        liga(L) = disp
        disp = x
        L = 0
        x = 0
fin(liberar_lista_circular)
```

Por consiguiente, debido a que el sub_programa de retornar lista es bastante más eficiente con listas circulares tomaremos la decisión de representar las listas ligadas como circulares cuando la operación de liberar lista tenga alta frecuencia.

Sin embargo, esta decisión afectará todos los otros algoritmos del manejo de la lista ligada y tocará evaluar si se justifica implantar esta representación cuando el proceso de liberar listas sea de alta frecuencia.

Cuando se recorre una lista ligada hay dos situaciones fundamentales que se deben controlar: una la situación de lista vacía y la otra la situación de terminación de recorrido de la lista.

Analizaremos el control de estas situaciones para los diferentes tipos de lista.

4.8 DIFERENTES TIPOS DE LISTAS LIGADAS Y SUS CARACTERÍSTICAS

Listas simplemente ligadas

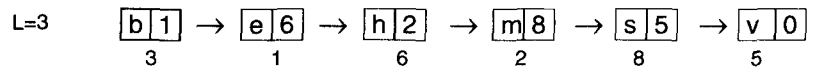


Figura 4.7

Recordemos nuestro algoritmo de recorrido.

```
sub_programa recorrer_lista(L)
      p = L
      while p <> 0  do
            write(dato(p))
            p = liga(p)
   end(while)
fin(recorrer_lista)
```

El algoritmo recorre e imprime el contenido de cada registro de la lista. Aquí se controla la situación de lista vacía, **L = 0**, con la misma instrucción con que se controla la terminación de recorrido de la lista: **while p <> 0**.

Es decir, con una sola instrucción estamos controlando dos situaciones.

Resumiendo, cuando tenemos una lista simplemente ligada, las características de lista vacía y recorrido son:

Lista vacía: L = 0
Primer registro: p = L
Ultimo registro: liga(p) = 0
Terminación de recorrido: p = 0

Lista simplemente ligada circular

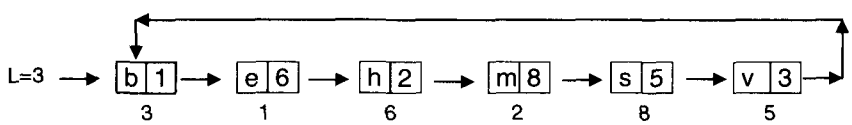


Figura 4.8

Para recorrer la lista no podemos elaborar un algoritmo como en el caso de la lista simplemente ligada porque cuando tenemos la lista circular ningún registro tendrá campo de liga cero (0), por tanto la situación de terminación de recorrido no se podrá controlar con la instrucción **while p <> 0.**

Cuando tenemos la lista circular el recorrido se termina cuando la variable auxiliar para el recorrido, es decir **p**, vuelva a ser igual a **L**. Plantear el ciclo con la instrucción **while p <> L** no sería correcto porque la primera instrucción es asignarle a la variable **p** el contenido de la variable **L**, y entonces al preguntar si **p** es diferente de **L** daría como resultado falso y nunca entraría al ciclo.

Por consiguiente debemos plantear un ciclo que permita ejecutar primero las instrucciones del ciclo y después preguntar por la condición de si **p** es diferente de **L**. Para ello disponemos de la instrucción de ciclo **do — while**. Nuestro algoritmo es:

```
sub_programa recorrer_lista_circular(L)
      p = L
      do
          write(dato(p))
          p = liga(p)
      while p <> L
fin(recorrer_lista_circular)
```

Lamentablemente este algoritmo no controla la situación de lista vacía.

En caso de que el parámetro **L** entre valiendo 0, la primera vez **p** queda valiendo 0, entra al ciclo y cuando vaya a ejecutar **write(dato(p))** nuestro programa cancela.

Para que el programa NO cancele debemos añadir instrucciones, tales como preguntar si **p <> 0**, antes del ciclo **do — while**, lo cual, implica que en todas las situaciones donde haya que recorrer una lista habrá que controlar dicha situación con más instrucciones, lo cual va en detrimento del programa.

¿Qué pasó entonces?: que por querer hacer eficiente la operación **liberar_lista**, estamos desmejorando todos los algoritmos en los cuales haya que recorrer alguna lista, porque necesitamos instrucciones diferentes para controlar las situaciones de lista vacía y de terminación de recorrido de la lista, además las operaciones de inserción y borrado tendrán orden de magnitud **O(n)**. Veamos por qué.

```
function buscar_donde_insertar(L, d)
    if L= 0 or d < dato(L)
       return(0)
    end(if)
    y = L
    p = liga(L)
    while p <> L and dato(p) < d do
         y = p
         p = liga(p)
    end(while)
    return(y)
fin(buscar_donde_insertar)
```

En el algoritmo anterior controlamos la situación de lista vacía y de comparación del primer dato con instrucciones adicionales. Nuestro algoritmo para insertar es:

```
subprograma insertar(L, y, d)
    new(x)
    dato(x) = d
    casos
        :L = 0:                    // inserta en lista vacía
              L = x
              liga(x) = x
        :y = 0:                    // inserta al principio
              u = ultimo(L)
              liga(x) = L
              liga(u) = x
              L = x
        :else:                     // inserta en sitio intermedio
              liga(x) = liga(y)
              liga(y) = x
    fin(casos)
fin(insertar)
```

En el anterior algoritmo cuando el registro a insertar vaya a quedar al principio habrá que buscar el último registro de la lista, algoritmo que tiene orden de magnitud **O(n)**, siendo **n** el número de registros de la lista. Este algoritmo lo presentamos a continuación.

```
function ultimo(L)
    if L = 0 then
        return(0)
```

```
    end(if)
    p = L
    while liga(p) <> L do
          p = liga(p)
    end(while)
    return(p)
fin(ultimo)
```

Nuestro objetivo es poder tener una lista en la cual liberar lista sea eficiente y además controlar las situaciones de lista vacía y de terminación de recorrido con una sola instrucción.

Dicho objetivo lo logramos añadiendo un registro al principio de la lista, el cual llamaremos **registro cabeza**.

Lista simplemente ligada circular con registro cabeza

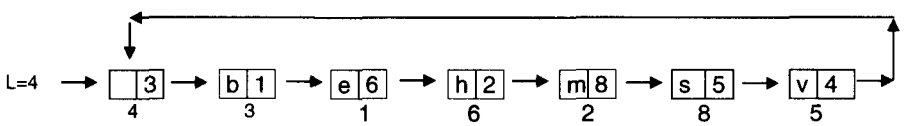


Figura 4.9

El registro cabeza es un registro que, por lo general, no tendrá información correspondiente al objeto que se esté representando en la lista ligada, **siempre** será el primer registro de la lista ligada y facilita todas las operaciones sobre la lista: construir la lista, recorrer la lista, insertar un registro, borrar un registro y liberar la lista.

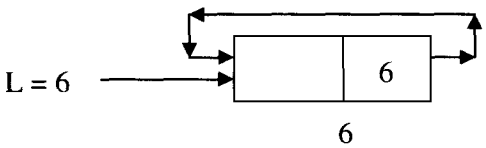
Realmente, si un registro adicional presenta tantas ventajas, no debemos preocuparnos mucho por el hecho de consumir una posición más de memoria.

El algoritmo de recorrido de la lista es:

```
sub_programa recorrer_lista(L)
      p = liga(L)
      while p <> L do
            write(dato(p))
            p = liga(p)
      end(while)
fin(recorrer_lista)
```

Para recorrer la lista se comienza en el segundo registro, es decir, a **p** se le asigna **liga(L)**. La lista se termina de recorrer cuando **p** sea igual a **L**.

La situación de lista vacía se representa así:



El campo de liga del registro cabeza apunta hacia sí mismo.

Por consiguiente, si la lista está vacía, inicialmente **p** queda valiendo **L** y no entra al ciclo de recorrido.

Con una sola condición estamos controlando dos situaciones: lista vacía y fin de recorrido.

Veamos también cómo se mejoran los algoritmos para las operaciones de inserción y borrado cuando tenemos la lista simplemente ligada circular con registro cabeza.

Consideremos primero el proceso de inserción.

```
function buscar_donde_insertar(L, d)
      y = L
      p = liga(L)
      while  p <> L and  dato(p) < d  do
            y = p
            p = liga(p)
      end(while)
      return(y)
fin(buscar_donde_insertar)

subprograma insertar(L, y, d)
      new(x)
      dato(x) = d
      liga(x) = liga(y)
      liga(y) = x
fin(insertar)
```

En nuestro subprograma para insertar no es necesario controlar ninguna situación especial ya que la variable **y** nunca será cero debido a que como tiene regis-

tro cabeza, en la función de buscar dónde insertar, su valor inicial es precisamente el registro cabeza y como la lista es circular el valor retornado siempre será diferente de cero.

Consideremos ahora el proceso de borrado

```
sub_programa buscar_dato_a_borrar(L, d, x, y)
     y = L
     x = liga(L)
     while x <> L and dato(p) <> d do
          y = x
          x = liga(x)
     end(while)
fin(buscar_dato_a_borrar)


sub_programa borrar(L,x,y)
     if x = L then
          write(«registro a borrar no existe»)
          return
      end(if)
      liga(y) = liga(x)
      free(x)
fin(borrar)
```

Nuevamente, por el hecho de tener registro cabeza nuestro algoritmo de borrado no requiere controlar situaciones especiales.

Lista simplemente ligada NO circular con registro cabeza

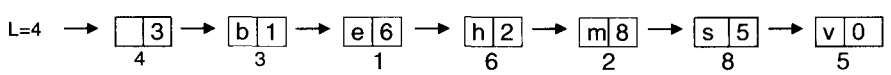


Figura 4.10

Los algoritmos para las operaciones básicas en esta nueva forma se dejan como ejercicio al lector.

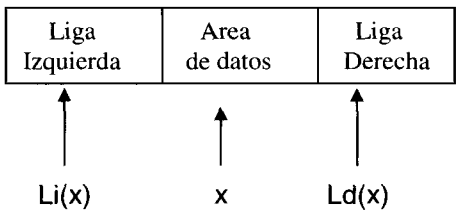
El caso es que, de acuerdo al problema en particular que se quiera desarrollar utilizando listas ligadas, se tomará la decisión de cuál tipo de representación utilizar.

4.9. LISTAS DOBLEMENTE LIGADAS

Hasta aquí hemos tratado con listas ligadas que sólo se pueden recorrer en un solo sentido: de izquierda a derecha.

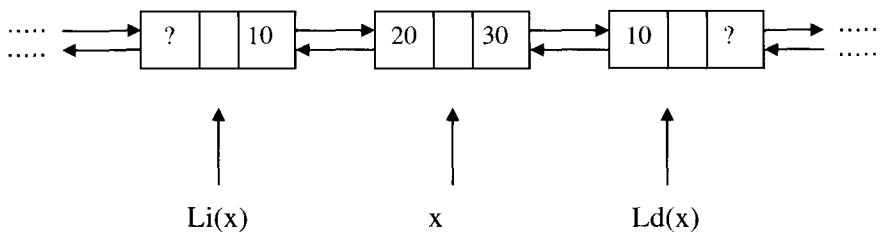
Existen cierto tipo de problemas que exigen que las listas ligadas se puedan recorrer en ambas direcciones: de izquierda a derecha y de derecha a izquierda.

Para lograr esto debemos diseñar registros (nodos) con dos campos de liga: uno que llamaremos **Li** y otro que llamaremos **Ld**. Un esquema general de dicho nodo es:



Li(x): apunta hacia el registro anterior a **x**.
Ld(x): apunta hacia el registro siguiente de **x**.

Consideremos el siguiente segmento de lista doblemente ligada para determinar lo que llamaremos la propiedad fundamental de las listas doblemente ligadas.



El registro anterior a **x** es el registro **Li(x)** y el registro siguiente a **x** es el registro **Ld(x)**. La propiedad fundamental de las listas doblemente ligadas es:

$$\mathbf{Ld(Li(x)) = x = Li(Ld(x))}$$

Lo cual, dicho en palabras es: el campo liga derecha del registro liga izquierda de **x** es el mismo registro **x**, y, el campo liga izquierda del registro liga derecha de **x** es el mismo registro **x**.

Al manejar listas doblemente ligadas, las operaciones básicas son las mismas que en las listas simplemente ligadas: recorrer la lista, insertar un registro y borrar un registro. Consideremos cada una de ellas.

Recorridos sobre la lista

En listas doblemente ligadas se tiene la propiedad de que la lista se puede recorrer en ambos sentidos: de izquierda a derecha y de derecha a izquierda. Veamos dichos recorridos.

Recorrido de izquierda a derecha

El algoritmo para recorrer una lista doblemente ligada de izquierda a derecha es idéntico al de recorrido sobre una lista simplemente ligada, la única diferencia es que para avanzar sobre la lista lo haremos utilizando el campo de liga derecha. Nuestro algoritmo es:

```
Sub_programa recorre_izq_der(L)
      p = L
      while  p <> 0  do
          write(dato(p))
          p = Ld(p)
      end(while)
fin(recorre_izq_der)
```

Recorrido de derecha a izquierda

Para recorrer la lista de derecha a izquierda debemos ubicarnos inicialmente en el último registro de la lista. Para ello utilizaremos una función, la cual llamaremos **ultimo,** y a partir de este registro nos desplazaremos sobre la lista utilizando el campo de liga izquierda. Nuestro algoritmo es:

```
Sub_programa recorre_der_izq(L)
      p = ultimo(L)
      while  p <> 0  do
          write(dato(p))
          p = Li(p)
      end(while)
fin(recorre_der_izq)
```

Consideremos entonces la función **ultimo** utilizada en dicho algoritmo. Para identificar el último registro en una lista doblemente ligada basta recorrer la lista buscando un registro cuyo campo de liga derecha sea cero. Nuestro algoritmo es:

```
function ultimo(L)
      if L = 0 then
          return(0)
      end(if)
      p = L
      while Ld(p) <> 0 do
          p = Ld(p)
      end(while)
      return(p)
fin(ultimo)
```

Inserción en una lista doblemente ligada

El proceso de inserción consta de dos partes: buscar dónde insertar e insertar.

Buscar dónde insertar

El subprograma buscar dónde insertar es idéntico al de las listas simplemente ligadas. Se recorre la lista comparando el dato de **p** con el dato a insertar y se avanza con el campo de liga derecha. Se retorna también el registro a continuación del cual hay que insertar un nuevo registro con dato **d**. Nuestro algoritmo es:

```
function buscar_donde_insertar(L, d)
      p = L
      y = 0
      while p <> 0 and dato(p) < d do
          y = p
          p = Ld(p)
      end(while)
      return(y)
fin(buscar_donde_insertar)
```

Insertar un registro x a continuación de un registro y

Consideremos la siguiente lista:

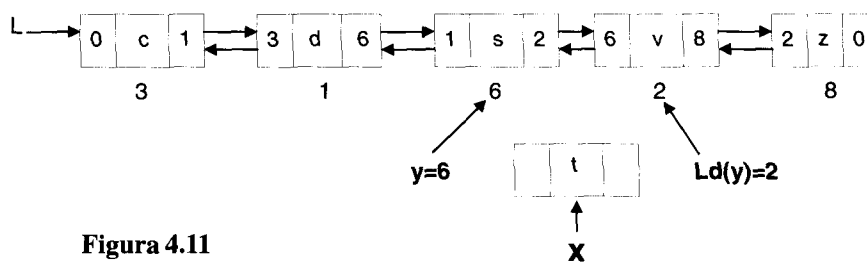


Figura 4.11

Si deseamos insertar la letra **"t"** en ella, primero ejecutamos buscar dónde insertar y dicho algoritmo retornará **y = 6**, es decir, debemos insertar un nuevo registro con el dato **"t"** a continuación del registro 6.

Las instrucciones para insertar el registro **x** a continuación del registro **y** en el ejemplo de la figura 4.11 son:

```
1. Ld(x) = Ld(y)
2. Li(x) = y
3. Li(Ld(x)) = x
4. Ld(y) = x
```

Figura 4.12

Las instrucciones 1 y 2 preparan el registro **x** para que funcione adecuadamente cuando se conecte a la lista. Las instrucciones 3 y 4 conectan el registro **x** con la lista.

Si el dato a insertar fuera a quedar de último la instrucción 3 fallaría, ya que estaríamos tratando de asignarle al registro **Ld(x)**, el cual vale cero, el registro **x**.

Por consiguiente, cuando el registro a insertar vaya a quedar de último las instrucciones son:

```
Li(x) = y
Ld(x) = 0
Ld(y) = x
```

Si el dato a insertar fuera a quedar de primero la instrucción 1 de la figura 4.12 fallaría, ya que el registro **y** es 0 y referirnos al campo **ld(y)** ocasiona un error.

Las instrucciones para insertar un registro al principio de la lista son:

```
1.  ld(x) = L
2.  if  L <> 0  then
3.  Li(L) = x
4.  end(if)
5.  L = x
```

En las instrucciones 2 y 3 se controla la eventualidad de que la lista **L** en la cual hay que efectuar la inserción esté vacía.

Considerando las tres situaciones presentadas, nuestro algoritmo para insertar un registro **x** a continuación de un registro **y** en una lista doblemente ligada es:

```
subprograma insertar(L, y, d)
   conseguir_registro(x)
   dato(x) = d
   Li(x) = y
   casos
      :y = 0:
           Ld(x) = L
           if L <> 0 then
              Li(L) = x
           end(if)
           L = x
      :Ld(y) = 0:
           Ld(x) = 0
           Ld(y) = x
      :else:
           Ld(x) = Ld(y)
           Li(ld(x)) = x
           Ld(y) = x
   fin(casos)
fin(subprograma)
```

Borrado de un registro de la lista doblemente ligada

El borrado de un registro consta también de los procesos buscar dato a borrar y borrar.

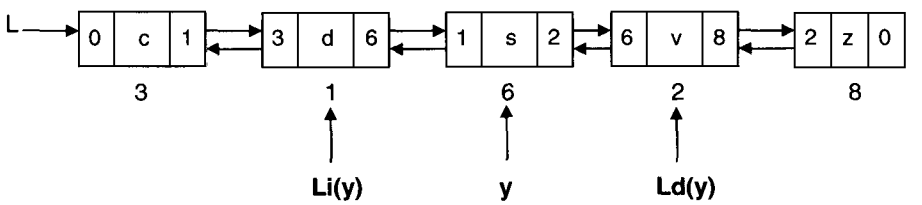
El proceso de buscar dato a borrar es más sencillo que en las listas simplemente ligadas, ya que para cada registro conocemos fácilmente el registro anterior y el registro siguiente. Veamos dicho algoritmo

```
Function buscar_dato(L, d)
      if  L = 0  then
           return(0)
      end(if)
      y = L
      while  y <> 0   and  dato(y) <> d  do
           y = Ld(y)
      end(while)
      return(y)
fin(buscar_dato)
```

Nuestro algoritmo **buscar_dato(L, d)** retorna **y**: dirección del registro a borrar. La variable **y** será cero si el dato buscado no se encuentra.

Analicemos ahora el proceso de borrado. Llamemos **y** el registro a borrar.

Recuerde que borrar un registro de una lista ligada, simplemente consiste en desconectar el registro de la lista.



Si queremos borrar el registro **y = 6**, aprovechando que la lista es doblemente ligada, tenemos que el registro anterior a **y** es el registro **Li(y), (Li(y)=1)** y el registro siguiente a **y** es el registro **Ld(y), (Ld(y)=2)**.

Para desconectar el registro **y** en sentido de izquierda a derecha lo que hay que hacer es que cuando estemos en el registro **Li(y)** debemos pasarnos hacia el registro **Ld(y),** o sea brincarnos el registro **y**. La instrucción para efectuar esta operación es:

Ld(Li(y))= Ld(y)

Si estamos recorriendo de derecha a izquierda, cuando estemos ubicados en el registro **Ld(y),** debemos pasarnos hacia el registro **Li(y),** o sea brincarnos el registro **y**. La instrucción para efectuar esta operación es:

Li(Ld(y)) = Li(y)

En este caso también debemos considerar varias situaciones especiales.

Si el registro a borrar es el último, es decir, **Ld(y) = 0**, la instrucción **Li(Ld(y)) = Li(y)** fallará ya que nos estaríamos refiriendo al campo liga izquierda del registro **Ld(y)**, el cual es cero.

Por consiguiente, cuando el registro a borrar sea el último la única instrucción para efectuar el borrado es:

Ld(Li(y)) = Ld(y)

Si el registro a borrar es el primero, es decir, **Li(y) = 0**, la instrucción **Ld(Li(y)) = Ld(y)** fallará, ya que el registro **Li(y)** es cero.

Cuando el registro a borrar sea el primero, el registro que debe quedar de primero es el que estaba de segundo, y la instrucción para lograr esto es **L = Ld(y)**. Y como el nuevo primero debe tener cero en el campo de liga izquierda, le asignamos cero al campo de liga izquierda de **L**: **Li(L) = 0**.

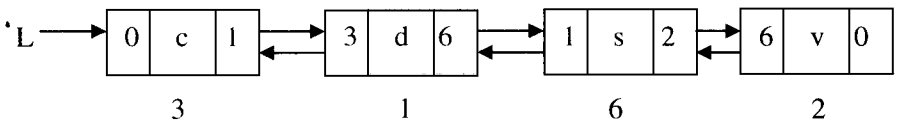
Sin embargo, como la lista pudo haber tenido sólo un registro, cuando ejecutamos la instrucción **L=Ld(L)**, **L** queda valiendo cero y la instrucción **Li(L)=0** fallará. Por consiguiente la instrucción **Li(L)=0** debemos condicionarla al hecho de que **L** sea diferente de cero.

Considerando estas situaciones, el algoritmo completo para borrar un registro de una lista doblemente ligada es:

```
sub_programa borrar (L, y)
      if  y = 0 then
            write('dato a borrar no existe')
            return
      end(if)
      casos
            :Li(y) = 0:                  // borra el primer registro
                  L = Ld(y)
                  if  L <> 0  then
                        Li(L) = 0
                  end(if)
            :Ld(y) = 0:                  // borra el ultimo registro
                  Ld(Li(y)) = 0
            :else:                       / borra registro intermedio
                  Li(Ld(y)) = Li(y)
                  Ld(Li(y)) = Ld(y)
      fin(casos)
      free(y)
fin(borrar)
```

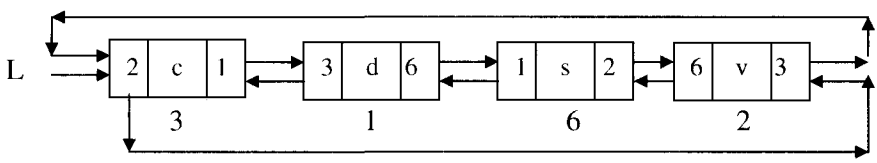
4.10 DIFERENTES TIPOS DE LISTAS DOBLEMENTE LIGADAS

Listas doblemente ligadas



El campo de liga izquierda del primer registro es cero porque el primer registro no tiene anterior, y el campo de liga derecha del último registro también vale cero porque el último registro no tiene siguiente. Los algoritmos para las operaciones básicas ya fueron presentadas con anterioridad.

Listas doblemente ligadas circulares



En las listas doblemente ligadas circulares el campo de liga derecha del último registro apunta hacia el primer registro de la lista y el campo de liga izquierda del primer registro apunta hacia el último registro de la lista.

Veamos cómo serían los algoritmos de recorridos, inserción y borrado para las listas doblemente ligadas circulares.

En las listas doblemente ligadas circulares se presentan los mismos problemas que se tenían con las lista simplemente ligadas circulares: las situaciones de lista vacía y de fin de recorrido se deben controlar con instrucciones diferentes.

Veamos primero los algoritmos para los recorridos

```
Sub_programa recorre_izq_der(L)
    if L = 0 then
        return
    end(if)
    p = L
    do
        write(dato(p))
```

```
        p = Ld(p)
    while p <> L
fin(recorre_izq_der)

Sub_programa recorre_der_izq(L)
    if L = 0 then
        return
    end(if)
    p = Li(L)
    do
        write(dato(p))
        p = Li(p)
    while p <> Li(L)
fin(recorre_der_izq)
```

Consideremos ahora los procesos de inserción y borrado en listas doblemente ligadas circulares.

Como hemos estado tratando, el proceso de inserción consta de dos partes: buscar dónde insertar e insertar. Nuestros algoritmos para ello son:

```
Function buscar_donde_insertar(L, d)
    if L = 0 then                    // controla situación de lista vacía
        return(0)
    end(if)
    if d < dato(L) then              // dato a insertar debe quedar de primero
        return(0)
    end(if)
    y = L
    p = Ld(L)
    while p <> L and dato(p) < d do
        y = p
        p = Ld(p)
    end(while)
    return(y)
fin(buscar_donde_insertar)
```

Nuestro algoritmo **buscar_donde_insertar(L,d)** retornará también un registro **y**, el cual apunta hacia el registro a continuación del cual hay que insertar un nuevo registro. La variable **y** valdrá cero si el registro a insertar debe quedar de primero o si la lista está vacía.

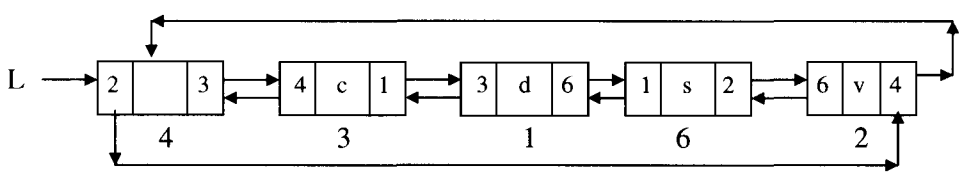
```
Subprograma insertar(L, y, d)
1.    new(x)
2.    dato(x) = d
3.    if  L = 0  then
4.          Li(x) = x
5.          Ld(x) = x
6.          L = x
7.          return
8.    end(if)
9.    sw = 0
10.   if  y = 0   then
11.         y = Li(L)
12.         sw = 1
13.   end(if)
14.   Ld(x) = Ld(y)
15.   Li(x) = y
16.   Li(Ld(x)) = x
17.   Ld(y) = x
18.   if  sw = 1  then
19.         L = x
20.   end(if)
fin(insertar)
```

Instrucciones 3 a 8 controlan la situación de lista vacía.

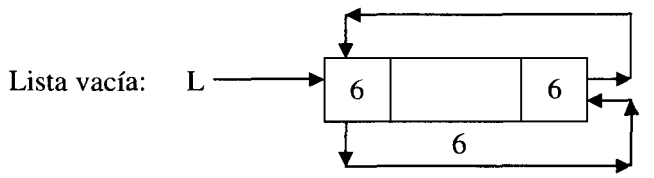
Instrucciones 9 a 13 controlan que el dato a insertar quede primero.

Dado que las instrucciones para insertar un registro en cualquier parte de la lista son las mismas, debemos controlar la situación en la cual hay que insertar el nuevo registro al principio de la lista (**y = 0**). Para ello manejamos un suiche (**sw**) el cual inicializamos en cero y lo convertimos en uno si el registro a insertar debe quedar de primero. En instrucciones 18 a 20 consideramos esta situación y actualizamos **L** si el registro a insertar será el primero.

Listas doblemente ligadas circulares con registro cabeza



En las listas doblemente ligadas circulares con registro cabeza se tiene la ventaja de que la representación de la lista vacía tiene como mínimo el registro cabeza:



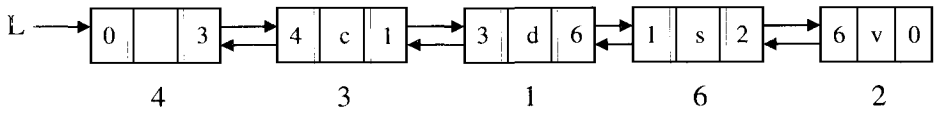
Veamos para esta nueva representación la forma como se facilitan y se hacen más eficientes los algoritmos para las operaciones básicas sobre listas ligadas.

```
Sub_programa recorre_izq_der(L)
      p = Ld(L)
      while  p <> L  do
            write(dato(p))
            p = Ld(p)
      end(while)
fin(recorre_izq_der)

Sub_programa recorre_der_izq(L)
      p = Li(L)
      while  p <> L   do
            write(dato(p))
            p = Li(L)
      end(while)
fin(recorre_der_izq)
Function buscar_donde_insertar(L, d)
      y = L
      p = Ld(L)
      while   p <> L  and  dato(p) < d   do
            y = p
            p = Ld(p)
      end(while)
      return(y)
fin(buscar_donde_insertar)

Subprograma insertar(L, y, d)
      new(x)
      dato(x) = d
      Ld(x) = Ld(y)
      Li(x) = y
```

```
        Li(Ld(x)) = x
        Ld(y) = x
fin(insertar)


function buscar_dato(L, d)
        p = Ld(L)
        while p <> L and dato(p) <> d do
                p = Ld(p)
        end(while)
        return(p)
fin(buscar_dato)


Sub_programa borrar(L,x)
        if x = L then
                write('dato no existe')
                return
        end(if)
        Ld(Li(x)) = Ld(x)
        Li(Ld(x)) = Li(x)
fin(borrar)
```

Dejamos al lector el ejercicio de confrontar los algoritmos desarrollados para cada uno de los diferentes tipos de lista.

Listas doblemente ligadas no circulares con registro cabeza



Dejamos al lector la tarea de desarrollar los algoritmos básicos para esta otra variación de lista doblemente ligada.

EJERCICIOS PROPUESTOS

1. Escriba un algoritmo que reverse los apuntadores de una lista simplemente ligada

2. Escriba un algoritmo que reverse los apuntadores de una lista simplemente ligada circular.

3. Escriba un algoritmo que intercambie los registro **x** e **y** de una lista ligada. Considere todos los tipos de lista y todos sus posibles casos.

4. Escriba algoritmos para ordenar listas ligadas en forma ascendente. Considere todos los tipos de lista.

5. Escriba algoritmos completos para determinar el registro que tenga el menor dato en una lista ligada. Considere todos los tipos de lista.

6. Escriba algoritmo para determinar el promedio de los datos de una lista ligada.

7. Escriba un algoritmo para insertar un registro **x** antes de un registro **y** en una lista doblemente ligada.

8. Escriba un algoritmo que dada una lista ligada, intercambie el primer registro con el último, el segundo con el penúltimo, el tercero con el antepenúltimo y así sucesivamente. Considere todos los tipos de lista.

9. Escriba algoritmo para determinar el promedio de los datos de una lista ligada. Considere todos los tipos de lista.

10. Se tiene una lista ligada con sus datos ordenados en forma ascendente. Pueden haber datos repetidos. Escriba un algoritmo que elimine los registros que tengan datos repetidos. Considere todos los tipos de lista.

5

ESTRUCTURAS DE DATOS Y SU DEFINICIÓN EN ABSTRACTO

5.1 INTRODUCCIÓN

Tipo de datos: se refiere a la clase de datos que se pueden almacenar en una variable en algún lenguaje de programación. Es decir, si una variable es de tipo entero los datos que se pueden almacenar en ella sólo podrán ser números enteros.

Datos objeto: es el conjunto de datos que conforman un determinado **Tipo**. Los datos objeto del tipo entero son todos los enteros desde - • hasta + • .

En una estructura de datos, se debe describir el conjunto de datos y la forma como ellos se relacionan. Esto último implica definir las operaciones que se pueden efectuar entre los datos objeto y la forma como ellos trabajan.

Las operaciones que se definan para una estructura de datos, las designaremos dentro del texto como FUNCIONES, y la definición o descripción de cómo ellas trabajan, las llamaremos AXIOMAS.

5.2 ESTRUCTURA NÚMEROS NATURALES

Empecemos con una estructura sencilla: la estructura números naturales:

```
ESTRUCTURA Num_nat.
    Funciones:
        Cero() → número_natural
        Suc(número_natural) → número_natural.  // SUC por SUCESOR
        Escero(número_natural) → lógico.
        Suma(número_natural, número_natural) → número_natural
        Igual(número_natural, número_natural) → lógico
```

Axiomas: $\forall x, y \in$ *número _ natural*

```
Escero(Cero) ::= verdad
Escero(Suc(x)) ::= falso
Suma(Cero, y) ::= y
Suma(Suc(x), y) ::= Suc(Suma(x, y))
Igual(Cero, x) ::= if Escero(x) then verdad
                   else falso
Igual(Suc(x), Cero) ::= falso
Igual(Suc(x), Suc(y)) ::= Igual(x, y)
```

Analicemos cuidadosamente el ejemplo anterior.

1. En la parte de axiomas se utiliza el símbolo **::=**, el cual se lee "SE DEFINE COMO" y se conoce como la notación BNF (Backus Naur Form).
2. En la parte correspondiente a las funciones se definen las operaciones que se pueden realizar, su sintaxis y el resultado que se obtiene al efectuar alguna de ellas.
3. En la definición de funciones siempre se debe incluir:
   a. Una función que cree la estructura vacía. En nuestro ejemplo es la función **Cero() → número_natural**, la cual crea la estructura números naturales vacía.
   b. Una función que genere o que permita incluir los elementos o datos objeto que conformarán la estructura a definir. En nuestro ejemplo es la función **SUCESOR (Suc)**.
4. La función **Cero()** crea la estructura números naturales vacía.
5. **Suc(Cero)** generará el primer número natural 1.
6. **Suc(Suc(Cero))** representa el número natural 2 y así sucesivamente.

En forma general si **Suc(x)** es un número natural **i**, **x** representa el número natural **i-1**.

En la parte de axiomas se describe la forma como trabajan las funciones. Se omiten en esta parte las dos funciones básicas, (creación y generación) definidas en el numeral 3, ya que su forma de operación está implícita en la definición de la función.

Aunque no hay una regla general para escribir axiomas intentaremos dar algunas normas para escribirlos: la definición de cómo opera una función consta la

mayoría de las veces de dos partes o axiomas: uno en la cual se hace referencia a la estructura vacía y otro en la cual se hace referencia a la estructura no vacía.

Para hacer referencia a la estructura vacía basta con invocar la función que crea la estructura vacía, y para hacer referencia a la estructura no vacía basta con invocar la función que genera o incluye elementos en la estructura.

La parte en la cual se hace referencia a la estructura no vacía, por lo general, se define con base en la misma función que se está definiendo.

Para ilustrar cómo operan dichos axiomas consideremos el ejemplo de sumar los números naturales 2 y 3.

El número natural 2 se representa:

```
Suc(Suc(Cero))
```

El número natural 3 se representa:

```
Suc(Suc(Suc(Cero)))
```

El planteamiento de la operación de suma según nuestro axioma es:

```
Suma(Suc(Suc(Cero)), Suc(Suc(Suc(Cero))))
```

De acuerdo con los axiomas definidos, debemos identificar en la operación planteada los términos que la conforman, es decir, identificar **x** e **y**:

- x = Suc(Suc(Cero))
- Suc(y) = Suc(Suc(Suc(Cero)))
- y = Suc(Suc(Cero))

teniendo identificados **x** e **y** procedemos a aplicar el axioma.

```
Suma(x, Suc(y)) ::= Suc(Suma(x, y))
```

y obtenemos lo siguiente:

```
Suc(Suma(Suc(Suc(Cero)),Suc(Suc(Cero))))
```

Nuevamente en el llamado a la ejecución de la función Suma, debemos identificar **x** y Suc(y).

- x = Suc(Suc(Cero))
- Suc(y)= Suc(Suc(Cero))
- y = Suc(Cero)

Ejecutamos Suma y obtenemos:

Suc(Suc(**Suma**(Suc(Suc(Cero))Suc(Cero))))

identificando nuevamente los términos **x** y Suc(y) en el llamado a la función Suma tenemos:

- x = Suc(Suc(Cero))
- Suc(y) = Suc(Cero)
- y = Cero

Ejecutamos Suma y obtenemos:

Suc(Suc(Suc(**Suma**(Suc(Suc(Cero)),Cero))))

identificando los términos que conforman el llamado a la función Suma, vemos que es de la forma Suma(x, Cero) cuyo resultado es **x**, por consiguiente, aplicando el axioma correspondiente obtenemos:

Suc(Suc(Suc(Suc(Suc(Cero)))))

Lo cual es la representación del número natural 5.

Dado el anterior ejemplo, vamos a definir más formalmente lo que es una estructura de datos.

Una estructura de datos es un conjunto de dominios ***D***, un dominio designado de ***D***, un conjunto de funciones ***F*** y un conjunto de axiomas ***A***. En el ejemplo anterior tenemos:

D = {números naturales, lógico}
d = Números naturales (dominio designado)
F = {Cero, Escero, Suc, Suma, Igual}

5.3 ESTRUCTURA ARREGLO

Un arreglo se define como un conjunto de pares índice y valor. Cada índice tiene un valor asociado a él. Definamos la estructura arreglo:

```
ESTRUCTURA Arreglo
  Funciones:
    Crear() → Arreglo
    Alm(Arreglo, Indice, Valor) → Arreglo        // Alm por Almacenar
    Esvacio(Arreglo) → Lógico
    Recuperar(Arreglo, Indice) → Valor
```

Axiomas:

Para todo A ∈ arreglos; i, J ∈ índices; x ∈ valores:

Esvacío(Crear) ::= Verdad
Esvacío(Alm(A, i, x)) ::= falso
Recuperar(Crear, J) ::= Error
Recuperar(Alm(A, i, x),J) ::= if i = J Then x
else Recuperar(A, J)

Analicemos la anterior estructura:

a. Se define la función Crear, la cual es una función constante que crea un arreglo vacío.
b. Se define la función Almacenar (Alm), la cual permite incluir los elementos que formarán parte de la estructura.

Cada elemento se representa con el par índice y valor, luego, para incluir un elemento nuevo en la estructura, se deben definir estos dos parámetros.

Esta definición no tiene ningún tipo de restricción, es decir, un mismo índice puede tener asociado más de un valor, diferentes índices pueden tener asociados el mismo valor, y los índices y valores pueden ser suministrados o incluidos en cualquier orden. Un ejemplo de arreglo podría ser el plano cartesiano. (Figura 5.1).

En dicho gráfico tenemos definidos los puntos (2,1), (4,2), (7,2) y (2,3).

Si deseamos incluir dichos puntos en nuestra estructura arreglo definida, basta con aplicar la función almacenar a cada pareja ***x, y*** que define cada uno de los puntos, teniendo presente que la primera vez que se aplique se haga sobre la estructura vacía Crear.

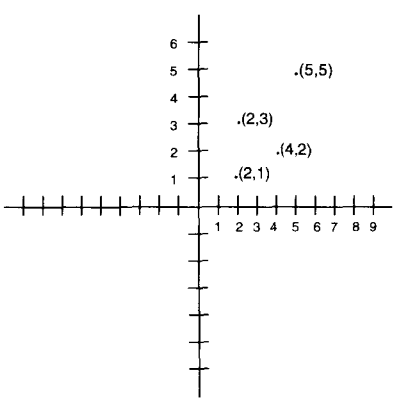


Figura 5.1

El punto (2,1) se incluirá así:

Alm(Crear,2,1)

La cual es la representación de un arreglo con un solo elemento: el índice 2 y el valor 1.

Si deseamos incluir el punto (4,2) en este arreglo, aplicamos nuevamente la función Alm:

Alm(Alm(Crear,2,1),4,2)

y tendríamos un arreglo con los dos elementos: (2,1) y (4,2)

La representación de los 4 puntos de nuestro ejemplo es:

Alm(Alm(Alm(Alm(Crear,2,1),4,2),5,5),2,3).

De acuerdo a la forma general que tenemos para representar un arreglo:

Alm(A, i, x),

en el arreglo definido con 4 puntos los parámetros **A**, **i** y **x** son:

A = Alm(Alm(Alm(Crear,2,1),4,2),5,5)
i= 2; x = 3

Es importante, para entender la forma como trabajan los axiomas, saber identificar los parámetros **A**, **i**, **x** en cualquier arreglo que se presente.

c. La función Esvacio determinará si un arreglo es vacío o no, y retornará un valor lógico: verdad o falso.
d. La función Recuperar permitirá obtener el valor asociado a un índice **J** cualquiera. Nótese que en caso de que en el arreglo que se representa, un mismo índice tiene más de un valor asociado, la función retornará el último valor que se incluyó en la estructura.

5.4 ESTRUCTURA LISTA ORDENADA

Una lista ordenada es un conjunto de elementos del mismo tipo que tienen cierta relación de orden. Algunos ejemplos son:

Los días de la semana:
(lunes, martes, miércoles, jueves, viernes, sábado, domingo)

Los meses del año:

(enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre, noviembre, diciembre)

Considerando una lista ordenada más abstractamente, diremos que: o puede estar vacía o se puede representar como:

$$(a_1, a_2, a_3, \ldots, a_n)$$

donde $\mathbf{a_i}$ son elementos de algún conjunto **S** y se denominan átomos.

Hay una variedad de operaciones que se pueden realizar sobre listas ordenadas. Algunas de ellas son:

1. Determinar la longitud **n** de la lista.
2. Recorrer la lista de izquierda a derecha o viceversa.
3. Recuperar el i-ésimo elemento de la lista ($1 \leq i \leq n$)
4. Almacenar un nuevo elemento en la i-ésima posición de la lista ($1 \leq i \leq n$).
5. Insertar un nuevo elemento en la posición **i** ($\mathbf{1 \leq i \leq n+1}$) tal que los elementos que ocupan las posiciones **i, i+1, i+2, ... ,n**, ocupen las posiciones **i+1, i+2, i+3, ..., n+1,** respectivamente.
6. Borrar el elemento de la posición **i** ($\mathbf{1 \leq i \leq n}$) de tal forma que los elementos que ocupan las posiciones **i+1, i+2, ... n**, ocupen las posiciones **i, i+1, ... , n-1** respectivamente.

A continuación definiremos, en abstracto, la estructura lista ordenada con sus funciones y axiomas:

ESTRUCTURA Lista_ordenada
funciones:

Crear() → Lista_ordenada
Alm(lista_ordenada, Entero, Atomo) → Lista_ordenada
Longitud(Lista_ordenada) → Entero
Recuperar(Lista_ordenada, Entero) → Atomo
Insertar(Lista_ordenada, Entero, Atomo) → Lista_ordenada
Borrar(Lista_ordenada, Entero) → Lista_ordenada.

Axiomas:
Para todo L ∈ Lista_ordenada,
i, j ∈ Enteros,
a, b, ∈ Atomos

```
Longitud(Crear) ::= 0
Longitud(Alm(L, i, a)) ::= 1 + Longitud(L)
Recuperar(Crear, j) ::= error
Recuperar(Alm(L, i, a), j) ::=  if i = j Then a
                                else Recuperar(L, j)
Insertar(Crear, j, b) ::= Alm(Crear, j, b)
Insertar(Alm(L, i, a), j, b) ::=  if i >= j Then
                                     Alm(Insertar(L, j, b), i+1, a)
                                  else
                                     Alm(Insertar(L, j, b), i, a)
Borrar(Crear, j) ::= Crear
Borrar(Alm(L, i, a), j) ::=  if i = j Then
                                Borrar(L, j)
                             else
                                if i > j Then
                                     Alm(Borrar(L, j), i – 1, a)
                                else
                                     Alm(Borrar(L, j), i, a)
```

Para explicar cómo funciona la anterior estructura consideremos la lista ordenada L = (a, b, e, h).

Una de las formas como podríamos escribir dicha lista según nuestras funciones es:

Alm(Alm(Alm(Alm(Crear,1,a),4,h),3,e),2,b)

es bueno resaltar que en dicha descripción los átomos de la lista se hallan físicamente desorganizados, sin embargo, ellos se hallan lógicamente ordenados por el entero, el cual desempeña la función de ordenador.

Las funciones Longitud y Recuperar no ofrecen mayor dificultad.

La función Borrar elimina el elemento asociado al entero **j** y continua auscultando en la lista por otro elemento que tenga ordenador **j**, hasta encontrar la lista vacía; además, los elementos cuyo ordenador **i** sea mayor o igual que **j**, se decrementan en uno.

La función Insertar incrementa en uno los ordenadores **i** que sean mayores o iguales a **j**, los que sean menores que **j** los conserva con su mismo valor. Al terminar de ejecutarse la función de inserción, el elemento definido por el ordenador **j** y el átomo **b** queda en la lista como si hubiera sido el primer elemento almacenado en ella.

La forma más común para representar una lista ordenada es utilizando arreglos, donde a cada átomo $\mathbf{a}_i$ de la lista se asocia el índice **i** del arreglo.

A esto nos referiremos como una correspondencia secuencial, debido a que en la representación convencional de arreglos, los elementos $\mathbf{a}_i$ y $\mathbf{a}_{i+1}$ se almacenan en las posiciones consecutivas **i** y **i+1** del arreglo. Esto nos permite recuperar y/o modificar elementos aleatorios del arreglo en una cantidad constante de tiempo, debido a que la memoria del computador tiene acceso aleatorio a cualquier palabra. Sólo las operaciones 5 y 6 requieren mayor costo. Inserción y borrado, utilizando almacenamiento secuencial, implica movimiento de elementos para conservar la correspondencia secuencial apropiadamente.

Es precisamente este sobrecosto, el que ha llevado a considerar formas de representación no secuenciales en el tratamiento de listas ordenadas.

5.5 ESTRUCTURA POLINOMIO

Un polinomio es una secuencia de términos de la forma $\mathbf{cx}^{\mathbf{e}}$, en la cual **c** se conoce como coeficiente y **e** como exponente.

Por ejemplo: $4x^3 + 5x^6 - 4x^3 + 2x$

Estamos interesados en definir una estructura polinomio con las operaciones que podamos realizar en ella.

```
Estructura Polinomio
 Funciones:
      Crear()→ Polinomio
      Añadir(Polinomio, coeficiente, exponente) → Polinomio
      Escero(Polinomio) → lógico
      Remover(Polinomio, exponente) → Polinomio
      Coef(Polinomio, exponente) → Coeficiente
      Suma(Polinomio, Polinomio) → Polinomio
      Smult(Polinomio, coeficiente, exponente) → Polinomio
      Mult(Polinomio, Polinomio) → Polinomio

Axiomas:
      Para todo P,Q ∈ Polinomios
                  c,d ∈ Coeficientes
                  e,f ∈ Exponentes

      Remover(Crear, f) ::= Crear
      Remover(Añadir(P, c, e), f) ::=
                        if  e = f  Then
```

```
                    Remover(P, f)
               else
                    Añadir(Remover(P, f), c, e)
Coef(Crear, f) ::= 0
Coef(Añadir(P, c, e), f) ::=
               if  e = f  Then
                    c + Coef(P, f)
               else
                    Coef(P, f)
Escero(Crear) ::= Verdad
Escero(Añadir(P, c, e)) ::=
               if  Coef(P, e) = -c  Then
                    Escero(Remover(P, e))
               else
                    Falso
Suma(Crear, Q) ::= Q
Suma(Añadir(P, c, e), Q) ::= Añadir(Suma(P, Q), c, e)
Smult(Crear, d, f) ::= Crear
Smult(Añadir(P, c, e), d, f) ::= Añadir(Smult(P, d, f), c*d, e+f)
Mult(Crear, Q) ::= Crear
Mult(Añadir(P, c, e), Q) ::= Suma(Mult(P, Q),Smult(Q, c, e))
```

Analicemos las funciones definidas y sus axiomas:

Crear, es la función constante que crea el polinomio vacío.
Añadir es la función que permite incluir un término en un polinomio.
Un término lo representamos simbólicamente por su coeficiente y su exponente.

Ejemplo: si tenemos el polinomio $4x^3 + 5x^6 - 4x^3 + 2x$, la forma como será descrito será así:

Añadir(Añadir(Añadir(Añadir(Crear,4,3),5,6),-4,3),2,1)

Note que no hemos hecho alguna restricción referente al orden en que se incluyan los términos ni a que puedan existir varios términos con el mismo exponente.

Los valores de los parámetros de acuerdo a la forma general son:

c = 2; d = 1;

P=Añadir(Añadir(Añadir(Crear,4,3),5,6),- 4,3)

La operación Remover elimina de un polinomio los términos que tengan un exponente dado.

La función Coef obtiene la suma de todos los coeficientes de los términos que tengan un exponente dado.

La función Escero determina si un polinomio es anulable o no.

Ejemplo: sea el polinomio:

$4x^3 + 2x^2 - 3x^3 - x^3 - 2x^2$ el cual es cero.

Su representación es:

Añadir(Añadir(Añadir(Añadir(Añadir(Crear,4,3),2,2),-3,3),-1,3),-2,2)

Antes de aplicar algún axioma se deben identificar los términos de acuerdo a la forma general de representar un polinomio, los cuales son **P**, **c**, **e**.

En nuestro ejemplo

c = -2
e = 2
P = Añadir(Añadir(Añadir(Añadir(Crear,4,3),2,2),-3,3),-1,3)

Al aplicar la función Coef, al polinomio **P**, el resultado que se obtiene es la suma de los coeficientes cuyo exponente es 2 (e=2), o sea, Coef(P)=2, lo cual significa que al agruparlo con el término representado por **c = -2 y e=2**, $(-2x^2)$ se cancelan los términos cuyo exponente es 2 (e=2):

Coef(P, e) = -c.

Por consiguiente la acción a tomar es eliminar del polinomio **P** los términos cuyo exponente sea 2 (e=2): Remover(P, e), y averiguar si el polinomio resultante es o no cero: Escero(Remover(P, e)).

El caso en que Coef(P, e) <> -c significa que los términos cuyo exponente es **e** no se cancelan, por tanto la función Escero arroja el resultado FALSO.

La función Suma prácticamente lo que hace es concatenar los polinomios **P** y **Q**, lo cual es realmente la suma de dos polinomios cuando todos los exponentes de los términos en **P** son diferentes de todos los exponentes de los términos en **Q**.

La función Smult multiplica un polinomio **P** por un término, el cual se define por su coeficiente **d** y su exponente **f**.

La función Mult, la cual multiplica dos polinomios, realmente se puede ver como la suma de multiplicar cada uno de los términos del polinomio **P**, por todos los términos del polinomio **Q**.

Estos axiomas son valiosos, ya que describen el significado de cada función de una manera concisa y sin implicar alguna forma de representación o implementación.

Si tomamos algunas decisiones como: los exponentes deben ser únicos y ser dados en orden decreciente. Esto simplifica considerablemente las operaciones Escero, Coef y Remover mientras que Suma, Smult y Mult permanecen igual. Consideremos entonces una nueva función Exp(polinomio), la cual retorna el grado de un polinomio (Grado = mayor exponente del polinomio) y escribamos una nueva versión del axioma Suma, la cual se puede ver más como un programa, pero es aún independiente de su representación.

C = A + B, donde A y B son los polinomios a sumar y C es el polinomio resultado.

```
C = Crear
while  Not Escero(A)  and  Not Escero(B)  do
     Casos
          :Exp(A) < Exp(B):
                C = Añadir(C, Coef(B, Exp(B)), Exp(B))
                B = Remover(B, Exp(B))
          :Exp(A) = Exp(B):
                C = Añadir(C, Coef(A, Exp(A)) + Coef(B, Exp(B)), Exp(A))
                A = Remover(A, Exp(A))
                B = Remover(B, Exp(B))
          :Exp(A) > Exp(B):
                A = Añadir(C, Coef(A, Exp(A)), Exp(A))
                B = Remover(A, Exp(A))
     fin(Casos)
end(while)
while  Not Escero(A)  do
        C = Añadir(C, Coef(A, Exp(A)), Exp(A))
        A = Remover(A, Exp(A))
end(while)
while  Not Escero (B)  do
        C = Añadir(C, Coef(B, Exp(B)), Exp(B))
        B = Remover(B, Exp(B))
end(while)
```

El ciclo básico de este algoritmo consiste en intercalar los términos de los dos polinomios, dependiendo del resultado de comparar sus exponentes. La instrucción CASOS determina la relación de los exponentes y ejecuta la instrucción apropiada. Cuando se termine un polinomio, se deben pasar los términos restantes del otro polinomio, que es lo que ejecutan los dos últimos ciclos.

EJERCICIOS PROPUESTOS

1. Defina la estructura Boolean con operaciones And, Or, Not, Imp, Eqv.
2. Defina una estructura Conjunto con operaciones Unión, Intersección, Pertenencia y Complemento.
3. Defina operaciones de multiplicación, resta, división, y exponenciación para la estructura números naturales.
4. Defina estructura String con operaciones Esvacio, Longitud, Concatenación, Subhilera y Posición.
5. Defina estructura Matriz con sus correspondientes funciones y axiomas.
6. Defina estructura Enteros con sus correspondientes funciones y axiomas.
7. Complete la estructura números naturales con las funciones módulo, máximo común divisor y mínimo común múltiplo.

5.6. REPRESENTACIÓN DE POLINOMIOS EN UN COMPUTADOR

Pasemos ahora a determinar cómo representar un polinomio en el computador. La forma general de un polinomio A(x) es:

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$

donde $\mathbf{a_n}$ es el coeficiente del término cuyo exponente es **n**. Decimos que el grado de **A** es **n**.

Para representar polinomios en un computador consideraremos siempre que los términos estarán ordenados descendentemente por exponente y que sólo podrá existir un término por exponente. Presentaremos tres diferentes formas para representarlos:

Representación de polinomios en vector forma 1

La forma 1 de representación de **A(x)** es una lista ordenada de coeficientes en un arreglo unidimensional de n + 2 elementos:

	1	2	3		n+1	n+2
A =	n	a_n	a_{n-1}		a_1	a_0

El primer elemento del vector **A** contendrá el grado del polinomio, los siguientes **n+1** elementos serán los coeficientes de los términos del polinomio en orden decreciente por exponente.

Ejemplo: sea el polinomio $4x^3 + 2x^3 - 6x + 1$
Su representación es:

1	2	3	4	5
3	4	2	-6	1

Si el polinomio fuera $8x^6 + 4x^3 - 5$ la representación es:

1	2	3	4	5	6	7	8
6	8	0	0	4	0	0	-5

En general un polinomio de grado **n** requiere un vector con **n + 2** elementos. O sea, que si el polinomio lo representamos en un vector llamado **A**, el último coeficiente se halla en la posición **A(1) + 2**.

Bajo esta representación interesará conocer cuál es el exponente asociado a un coeficiente ubicado en alguna posición del vector. Llamando **Pos** esta posición y **n** el grado del polinomio:

$$Exp = n - Pos + 2 \qquad (1)$$

La operación opuesta también es de interés: conocido un exponente interesará conocer la posición del vector en la cual se halla ubicado su coeficiente:

$$Pos = n - Exp + 2 \qquad (2)$$

Con esta representación, los algoritmos que resultan para manipular polinomios son sencillos. El polinomio vacío se representa con un -1 en la primera posición.

Pasemos a considerar algunos ejemplos para ilustrar la utilización de las fórmulas (1) y (2). Consideremos el algoritmo de sumar dos polinomios representados en vector forma 1. Presentaremos dos algoritmos: en el primero recorreremos los polinomios desde el término con exponente cero hasta el grado del polinomio, y en el segundo recorreremos los polinomios desde el término con mayor exponente hasta el término con exponente cero.

Consideremos el primero de ellos:

```
Sub_programa suma_pol(A, B, C)
1.    m = A(1) + 2
2.    n = B(1) +2
3.    C(1) = mayor(A(1), B(1))
4.    p = C(1) + 2
5.    while  m > 2  and  n > 2  do
6.          C(p) = A(m) + B(n)
7.          m = m – 1
8.          n = n – 1
9.          p = p – 1
10.   end(while)
11.   while  m > 2  do
12.         C(p) = A(m)
13.         m = m – 1
14.         p = p – 1
15.   end(while)
16.   while  n > 2  do
17.         C(p) = B(n)
18.         n = n – 1
19.         p = p – 1
20.   end(while)
21.   ajuste(C)
fin(suma_pol)
```

En el anterior algoritmo, en las instrucciones 1 y 2, ubicamos **m** y **n** en el extremo derecho de cada vector, para recorrerlos de derecha a izquierda e ir sumando los coeficientes de cada polinomio respectivamente; en la línea 3, utilizamos una función llamada **mayor(a,b)** la cual retorna el mayor de dos datos enviados como parámetros, y en la línea 4 nos ubicamos en el extremo derecho del vector **C**, el cual contendrá el resultado de la suma.

Instrucciones 5 a 10 suman los coeficientes de los términos que tienen igual exponente, y en instrucciones 11 a 15 y 16 a 20 terminamos de trasladar los coeficientes de los términos del polinomio que tenía mayor grado respectivamente.

Por último, en la instrucción 21 invocamos un subprograma llamado **ajuste(C)**, el cual simplificará el polinomio resultado en caso de que los primeros términos hayan resultado con coeficiente cero (0). Este algoritmo lo trataremos más adelante.

Veamos ahora, la segunda versión para sumar dos polinomios bajo esta representación.

```
Sub_programa suma_pol(A, B, C)
1.    m = A(1) + 2
2.    n = B(1) + 2
3.    C(1) = mayor(A(1), B(1))
4.    i = 2
5.    j = 2
6.    k = 1
7.    while i <= m do
8.          ea = A(1) – i + 2
9.          eb = B(1) – j + 2
10.         k = k + 1
11.         casos
12.               :ea > eb:
13.                     C(k) = A(i)
14.                     i = i + 1
15.               :ea < eb:
16.                     C(k) = B(j)
17.                     j = j + 1
18.               :ea = eb:
19.                     C(k) = A(i) + B(j)
20.                     i = i + 1
21.                     j = j + 1
22.         fin(casos)
23.   end(while)
24.   ajuste(C)
fin(suma_pol)
```

En este algoritmo recorremos los polinomios de izquierda a derecha comparando los exponentes de cada polinomio, los cuales determinamos con la fórmula (1). El algoritmo traslada el coeficiente del término con mayor exponente hacia el vector resultado hasta que los exponentes se igualan. A partir de este sitio los exponentes son iguales y termina de recorrer los vectores a sumar simultáneamente. Es por esto que la condición de terminación del ciclo (instrucción 7) sólo se controla con una variable.

Veamos ahora nuestro algoritmo ajuste(C).

Si sumamos dos polinomios como:

$$A = 5x^8 - 7x^7 + 9x^6 + 3x^5 + 2x^4 + 8x - 7$$
$$B = -5x^8 + 7x^7 - 9x^6 + 2x^5 - 3x^4 + 2x^3 - 5x^2 + x + 1$$

El resultado es:

$$C = 0x^8 + 0x^7 + 0x^6 + 5x^5 - x^4 + 2x^3 - 5x^2 + 9x - 6$$

Y su representación en vector forma 1 es:

1	2	3	4	5	6	7	8	9	10
8	0	0	0	5	-1	2	-5	9	-6

Como se puede ver, esta representación informa que el polinomio es de grado 8, pero los tres primeros términos tienen coeficiente 0, lo cual implica que realmente el polinomio es de grado 5. Por consiguiente debemos eliminar los tres primeros términos y colocar el grado del polinomio en 5. Nuestro algoritmo **ajuste(C)** efectúa esta tarea.

```
Sub_programa ajuste(C)
        i = 2
        n = C(1) + 2
        while  C(i) = 0  do
                i = i + 1
        end(while)
        for  k = i  to  n  do
                C(k – i + 2) = C(k)
        end(for)
        C(1) = C(1) – (n – i)
fin(ajuste)
```

Consideremos ahora un algoritmo para multiplicar dos polinomios bajo esta representación.

```
Sub_programa  mult_pol(A,B,C)
1.      C(1) = A(1) + B(1)
2.      for  i = 2 to C(1) + 2  do
3.              C(i) = 0
4.      end(for)
5.      m = A(1) + 2
```

```
6.    n = B(1) + 2
7.    for i = 2 to m do
8.          ea = A(1) – i + 2
9.          for j = 2 to n do
10.               eb = B(1) – j + 2
11.               ec = ea + eb
12.               k = C(1) – ec + 2
13.               C(k) = C(k) + A(i) * B(j)
14.         end(for)
15.   end(for)
fin(mult_pol)
```

Instrucción 1 determina el grado del polinomio resultado.

Instrucciones 2 a 4 inicializan los coeficientes del polinomio resultado en cero.

Instrucciones 5 y 6 determinan el límite de recorrido de cada vector.

Instrucciones 8 y 10 determinan los exponentes correspondientes a las posiciones **i** y **j** de los términos a multiplicar, con base en la fórmula (1)

Instrucción 11 determina el exponente del término resultante.

Instrucción 12 determina la posición en la cual debemos acumular el producto de los coeficientes de las posiciones **i** y **j** de los polinomios que se están multiplicando.

En el anterior algoritmo se pueden simplificar las instrucciones 8, 10, 11 y 12. Veamos cómo: la instrucción 12 es:

$$K = C(1) - ec + 2$$

Reemplazando **C(1)** por el lado derecho de la instrucción 1 y **ec** por el lado derecho de la instrucción 11 obtenemos:

$$K = A(1) + B(1) - (ea + eb) + 2$$

Y reemplazando aquí **ea** y **eb** por los lados derechos de las instrucciones 8 y 10 tendremos:

$$K = A(1) + B(1) - (A(1) - i + 2 + B(1) - j + 2) + 2$$

La cual, al simplificarla queda:

$$K = i + j - 2$$

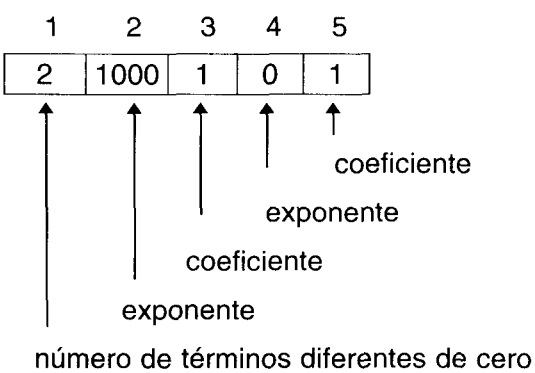
Aplicando esta simplificación nuestro algoritmo queda:

```
Sub_programa mult_pol(A,B,C)
1.    C(1) = A(1) + B(1)
2.    for  i = 2 to C(1) + 2  do
3.          C(i) = 0
4.    end(for)
5.    m = A(1) + 2
6.    n = B(1) + 2
7.    for  i = 2 to m  do
8.          for  j = 2 to n  do
9.                k = i + j – 2
10.               C(k) = C(k) + A(i) * B(j)
11.         end(for)
12.   end(for)
fin(mult_pol)
```

Representación de polinomios en vector forma 2

La representación de polinomios en vector forma 1 tiene algunas desventajas. Consideremos el polinomio $x^{1000} + 1$. Este polinomio requiere un vector de 1002 elementos, de los cuales 999 serán cero, lo cual constituye un desperdicio o una mala utilización de la memoria. Lo anterior ha llevado a plantear diferentes alternativas de representación. Una de ellas consiste en representar únicamente los términos diferentes de cero. Para representar un término se utilizarán dos posiciones del vector, una para el exponente y otra para el coeficiente, por tanto si un polinomio tiene **n** elementos diferentes de cero, se necesitarán **2*n** posiciones. Adicionalmente utilizaremos la primera posición para determinar el número de términos diferentes de cero. Por tanto, el total de elementos requeridos para el vector será **2*n+1**. Los términos se almacenan en orden decreciente por exponente.

Dado lo anterior, el polinomio $x^{1000} + 1$ se representa así:



Lo cual representa un ahorro de 997 posiciones de memoria.
Comparemos ahora las dos formas para representar polinomios.

Considerando el caso extremo del polinomio $x^{1000} + 1$, con la segunda forma de representación hemos logrado un ahorro de casi 200 veces los elementos que se necesitarían para representar dicho polinomio en la primera forma.

Veamos entonces el caso de cuando tengamos un polinomio de grado **n** con **n+1** términos, es decir, un polinomio lleno. Representemos el polinomio

$4x^3 + 2x^2 - 6x + 1$

en la segunda forma.

Su representación es:

1	2	3	4	5	6	7	8	9
4	3	4	2	2	1	-6	0	1

En la primera forma requerimos 5 elementos, y en la segunda nueve elementos, o sea menos del doble de los elementos necesitados en la primera forma. Por consiguiente, de la confrontación de estos casos extremos vemos que el ahorro llega a ser más considerable para casos extremos de términos iguales a cero que para casos extremos con todos los términos diferentes de cero, por consiguiente nos decidimos por la segunda forma de representación de polinomios.

Ya hemos definido la forma como vamos a representar los polinomios. Lo que se sigue es desarrollar "Software" para manipular los polinomios bajo esta representación.

Escribamos entonces un subprograma que sume polinomios representados de la segunda forma:

```
sub_programa suma_pol(A, B, C)
m = A(1)
      n = B(1)
      i = 2
      j = 2
      k = 2
      while i <= 2*m and j <= 2*n do
            casos
                  :A(i) = B(j):
                        C(k+1) = A(i+1) + B(j+1)
```

```
                        if  C(k+1) <>0  then
                              C(k) = A(i)
                              k = k + 2
                        end(if)
                        i = i + 2
                        j = j + 2
                  :A(i) < B(j):
                        C(k) = B(j)
                        C(k+1) = B(j+1)
                        j = j + 2
                        k = k + 2
                  :A(i) > B(j)
                        C(k) = A(i)
                        C(k+1) = A(i+1)
                        i = i + 2
                        k = k + 2
            fin(casos)
      end(while)
      while  i <= 2*m  do
            C(k) = A(i)
            C(k+1) = A(i+1)
            i = i + 2
            k = k + 2
      end(while)
      while  j <= 2*n  do
            C(k) = B(j)
            C(k+1) = B(j+1)
            j = j + 2
            k = k + 2
      end(while)
      c(1) = k/2 - 1
fin(suma_pol)
```

El procedimiento tiene parámetros los cuales son nombres de arreglos que representan polinomios. Se manejan tres variables **i, j, k** para movernos sobre los arreglos A, B y C respectivamente.

Analicemos el orden de magnitud de este algoritmo: es obvio hacer este análisis en función de **m** y **n**, el número de términos diferentes de cero en A y B respectivamente.

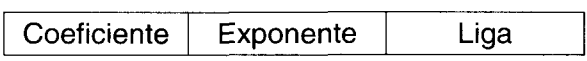
Las instrucciones de asignación de las primeras 5 líneas se ejecutan sólo una vez, por tanto contribuyen en O(1) al orden de magnitud total del procedimiento.

En caso de que **m = 0** ó **n = 0**, el primer ciclo While no se ejecuta. En caso en que ambos **m** y **n** sean diferentes de cero, se entra al ciclo. En cada iteración o se incrementa **i** en 2 o se incrementa **j** en 2, o ambos. El ciclo termina cuando **i** o **j** excedan los valores de **2*m** ó **2*n** respectivamente.

La situación más desfavorable se presenta cuando todos los exponentes de los términos del polinomio **A** son diferentes de todos los exponentes de los términos del polinomio **B** y el número de elementos diferentes de cero, es el mismo, o sea que **A(i)** siempre es diferente de **B(j)**, por tanto en cada iteración sólo **i** o **j** se incrementa en 2, lo que hace que el ciclo se ejecute **m+n-1** veces y su orden de magnitud es **O(m+n).** Los dos ciclos siguientes en el peor de los casos serán **O(m)** u **O(n)** respectivamente. Tomando la suma de todos estos tiempos obtenemos que el orden de magnitud de este algoritmo es **O(m+n)**.

Manejo de polinomios como listas ligadas

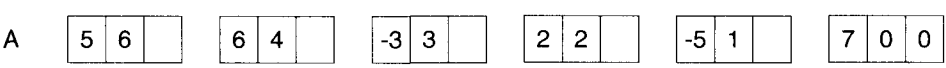
Como un polinomio es una sucesión de términos de la forma $\mathbf{cx^e}$, lo representaremos como lista ligada utilizando un nodo para cada término del polinomio, dichos registros los tendremos ordenados descendentemente por el campo de exponente. La configuración de cada registro para representar el polinomio como lista ligada es:



Dado el siguiente polinomio,

$$\mathbf{5x^6 + 6x^4 - 3x^3 + 2x^2 - 5x + 7}$$

su representación como lista ligada es:

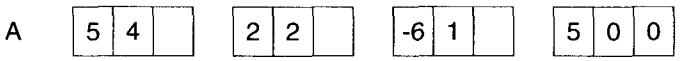


Los registros están ordenados descendentemente por campo de exponente.

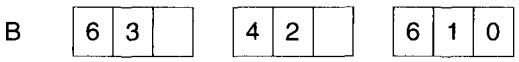
Teniendo definida la representación, el paso siguiente es desarrollar los programas para manipular el objeto con dicha representación. Trataremos aquí el algoritmo para sumar polinomios.

Consideremos los siguientes polinomios **A** y **B**

$A = 5x^4 + 2x^2 - 6x + 5$



$B = 6x^3 + 4x^2 + 6x$



Nota: como la forma de construir la lista resultado es agregando un registro al final de ella usamos la variable auxiliar **ultimo** la cual indica permanentemente cuál es el último registro. Usamos además también el registro auxiliar al principio de la lista y que al final del proceso lo borramos.

```
sub_programa suma_pol(A, B, C)
    p = A
    q = B
    new(C)
    ultimo = c
    while p <> 0 and q <> 0 do
        casos
            :exp(p) > exp(q):
                añadir_termino(coef(p), exp(p), ultimo)
                p = liga(p)
            :exp(p) < exp(q):
                añadir_termino(coef(q), exp(q), ultimo)
                q = liga(q)
            :exp(p) = exp(q):
                co = coef(p) + coef(q)
                if co <> 0 then
                    añadir_termino(co, exp(q), ultimo)
                end(if)
                p = liga(p)
                q = liga(q)
        fin(casos)
    end(while)
    while p <> 0 do
        añadir_termino(coef(p),exp(p),ultimo)
        p = liga(p)
    end(while)
    while q <> 0 do
        añadir_termino(coef(q),exp(q),ultimo)
        q = liga(q)
```

```
        end(while)
        liga(ultimo) = 0
        x = C
        C = liga(C)
        free(x)
fin(suma_pol)


sub_programa añadir_termino(co,e,u)
        new(x)
        coef(x) = co
        exp(x) = e
        liga(u) = x
        u = x
fin(sub_programa).
```

Si consideramos que sólo trabajaremos con polinomios cuyo exponente sea mayor o igual que cero, y utilizamos listas simplemente ligadas con registro cabeza, podremos obtener mejores algoritmos llenando el campo de exponente del registro cabeza con un exponente negativo.

El siguiente algoritmo suma dos polinomios **A** y **B** manejando listas circulares con registro cabeza. El resultado se almacena en una tercera lista **C**.

```
sub_programa suma_pol(A, B, C)
        p= liga(A)
        q= liga(B)
        new(C)
        ultimo = C
        while  p <> A  or  q <> B  do
                casos
                    :exp(p) > exp(q):
                                añadir_termino(coef(p), exp(p), ultimo)
                                p = liga(p)
                    :exp(p) < exp(q):
                                añadir_termino(coef(q), exp(q), ultimo)
                                q = liga(q)
                    :exp(p) = exp(q):
                                co = coef(p) + coef(q)
                                if co <> 0 then
                                        añadir_termino(co, exp(p), ultimo)
                                end(if)
                                p = liga(p)
```

```
                    q = liga(q)
            fin(casos)
        end(while)
        liga(ultimo) = C
        exp(C) = -1
fin(suma_pol).
```

Dejamos al lector el análisis del algoritmo anterior, y su comparación frente al inicialmente dado.

En general, cuando se representa un objeto como lista ligada con registro cabeza es, conveniente utilizar los campos del registro cabeza con información que facilite o mejore los algoritmos para manipular dicho objeto. Dicha información dependerá de cada problema en particular.

EJERCICIOS PROPUESTOS

1. Escriba algoritmos que ejecuten todas las operaciones sobre polinomios representados en un vector en la forma 1.

2. Escriba algoritmos que ejecuten todas las operaciones sobre polinomios representados en un vector en la forma 2.

3. Escriba algoritmos que ejecuten todas las operaciones sobre polinomios representado como listas ligadas. Considere todos los tipos de listas.

4. Escriba una algoritmo que sume dos polinomios P1 y P2 representados como listas ligadas. Su algoritmo no debe obtener memoria adicional y el resultado debe quedar en la lista P1, la lista P2 desaparece. Considere todos los tipos de lista.

6

MATRICES DISPERSAS

6.1 INTRODUCCIÓN

Una matriz es un objeto matemático que se presenta en una gran variedad de problemas. En general, una matriz consiste de un arreglo con **m** filas y **n** columnas de datos.

mat	1	2	3	4	5	6
1	1	0	0	2	0	-3
2	0	4	8	0	0	0
3	0	0	0	7	0	0
4	0	0	0	0	0	0
5	6	0	0	0	0	0
6	0	0	5	0	0	0

Figura 6.1

Tradicionalmente, una matriz se almacena en un arreglo de dos dimensiones. Cualquier elemento se puede trabajar designándolo por **mat(i,j)** donde **i** representa la fila y **j** la columna, y su acceso es muy rápido.

Si examinamos la matriz de la figura 6.1 (la cual llamaremos **mat**), vemos que muchos de sus elementos son cero. Tal matriz se conoce como dispersa.

No existe una norma precisa para decir cuándo una matriz es dispersa, es un concepto intuitivo. En nuestro ejemplo, de 36 elementos, sólo 8 son diferentes de cero y esa es una matriz dispersa.

Las matrices dispersas han motivado a considerar una forma alterna de representación, esto debido a que en la práctica se presentan matrices muy grandes, las

cuales son dispersas. Por ejemplo, una matriz de 1000*1000, con sólo 1000 elementos diferentes de cero, aún en los computadores modernos es traumático almacenar en memoria al mismo tiempo dicha matriz completamente.

Las representaciones alternas almacenarán únicamente los elementos diferentes de cero.

En general, existen dos formas para representar matrices dispersas: por extensión y por compresión. Por extensión consideraremos tres formas de hacerlo: en tripletas, como lista ligada forma 1 y como lista ligada forma 2, y por compresión, utilizando fórmulas de direccionamiento.

6.2 REPRESENTACIÓN DE MATRICES DISPERSAS EN TRIPLETAS

Cada elemento de la matriz está definido por sus posiciones **i** y **j,** las cuales representan la fila y la columna respectivamente, y para identificar completamente un elemento cualquiera debemos hacer referencia a la fila, la columna y al valor almacenado en dichas coordenadas.

Podemos almacenar la matriz como una lista de tripletas: **i, j, valor**

Las tripletas se representarán ordenadas ascendentemente por filas y dentro de fila por columnas. Adicionalmente utilizaremos una primera tripleta, la cual define el orden de la matriz y el número de elementos diferentes de cero.

La matriz de la figura 6.1 quedará:

A	1	2	3
1	6	6	8
2	1	1	1
3	1	4	2
4	1	6	-3
5	2	2	4
6	2	3	8
7	3	4	7
8	5	1	6
9	6	3	5
10	7	7	

Figura 6.2

Y la llamaremos **A**.

O sea, para representar una matriz dispersa utilizamos otra matriz, la cual tendrá 3 columnas y tantas filas como elementos diferentes de cero haya, más uno.

La primera fila nos da la información correspondiente al número de filas y columnas que tenga la matriz dispersa y al número de elementos diferentes de cero, las filas siguientes definen los elementos diferentes de cero ordenados por filas y dentro de fila por columnas, y al final añadiremos una tripleta, la cual nos facilitará la elaboración de muchos algoritmos, y cuya información será el número de filas más uno y el número de columnas más uno en los campos de fila y columna. El campo de valor de esta última tripleta podrá tener cualquier dato.

Después de decidir sobre una forma de representación, interesa desarrollar los algoritmos para trabajar sobre dicha representación. En el caso de matrices, algunas operaciones sobre ellas son: construir **A** conocido **mat**, construir **mat** conocido **A**, insertar una tripleta en una matriz de tripletas ya construida, borrar una tripleta, intercambiar dos filas, intercambiar dos columnas, construir la traspuesta, sumar matrices, multiplicar matrices, construir la inversa, etc.

Estudiaremos algunas de estas operaciones con nuestra forma de representación.

Comenzaremos elaborando algoritmos que nos familiaricen con la nueva forma de representación.

Construir A conocido mat.

```
subprograma construye_A(mat, m, n, A)
1.    A(1,1) = m
2.    A(1,2) = n
3.    A(1,3) = 1
4.    for  i = 1 to m  do
5.         for  j = 1  to  n do
6.              if  mat( i,j) <> 0  then
7.                   A(1,3) = A(1,3) + 1
8.                   A(A(1,3), 1) = i
9.                   A(A(1,3), 2) = j
10.                  A(A(1,3), 3) = mat(i,j)
11.             end(if)
12.        end(for)
```

```
13.   end(for)
14.   A(A(1,3)+1, 1) = m + 1
15.   A(A(1,3)+1, 2) = n + 1
16.   A(1,3) = A(1,3) - 1
fin(construye_A)
```

En el anterior subprograma los parámetros de entrada son **mat**, **m** y **n**, y **A** es un parámetro de salida. **A** es la matriz de tripletas en la cual se representa la matriz **mat**.

Instrucciones 1 a 3 conforman la primera tripleta con los datos del número de filas y número de columnas de la matriz a representar. El número de elementos diferentes de cero se inicializa en uno y se utilizará durante el algoritmo, en las instrucciones del ciclo, para controlar la tripleta en la cual debe quedar cada elemento diferente de cero de la matriz **mat**.

Construir MAT conocido A

```
Subprograma construye_mat(A, m, n, mat)
1.    m = A(1,1)
2.    n = A(1,2)
3.     for  i = 1 to m  do
4.        for  j = 1  to n  do
5.            mat(i,j) = 0
6.         end(for)
7.      end(for)
8.    p = A(1,3) + 1
9.    for  i = 2  to  p  do
10.       mat(A(i,1), A(i,2)) = A(i,3)
11.   end(for)
fin(construye_mat)
```

En el anterior algoritmo **A** es el parámetro de entrada y **m**, **n**, y **mat** son parámetros de salida.
En instrucciones 1 y 2 se determinan los valores de **m** y **n**.
En instrucciones 3 a 7 se inicializa la matriz **MAT** con ceros.
En instrucciones 8 a 11 se llevan los elementos diferentes de cero a sus respectivas posiciones en la matriz **MAT**.

Insertar una tripleta en una matriz de tripletas ya construida.

```
Subprograma insertar(A, f, c, v)
1.    if f > A(1,1) or f < 1 or c > A(1,2) or c < 1 then
2.        write('tripleta inválida')
3.        return
4.       end(if)
5.     i = 2
6.    while A(i,1) < f do
7.        i = i + 1
8.    end(while)
9.    while A(i,1) = f and A(i,2) < c do
10.       i = i + 1
11.   end(while)
12.   if A(i,1) = f and A(i,2) = c then
13.       write('Tripleta ya existe')
14.       return
15.   end(if)
16.   p = A(1,3) + 2
17.   while p >= i do
18.       A(p+1, 1) = A(p,1)
19.       A(p+1, 2) = A(p,2)
20.       A(p+1, 3) = A(p,3)
21.       p = p – 1
22.   end(while)
23.   A(i,1) = f
24.   A(i,2) = c
25.   A(i,3) = v
26.   A(1,3) = A(1,3) + 1
fin(insertar)
```

- Instrucciones 1 a 4 validan que los datos de fila y columna de la tripleta a insertar estén dentro del rango de la matriz.
- Instrucciones 5 a 8 posicionan **i** en la tripleta a partir de la cual se comienzan a representar los elementos diferentes de cero de la fila **f**.
- Instrucciones 9 a 11 determinan la posición en la cual deberá quedar la nueva tripleta.
- Instrucciones 12 a 14 controlan que la tripleta a insertar no vaya a quedar duplicada.
- Instrucciones 16 a 22 desplazan las tripletas, desde la tripleta **i** hasta la tripleta **p** (la última), una posición hacia abajo.
- Instrucciones 23 a 25 insertan la nueva tripleta.
- Instrucción 26 actualiza el número de tripletas en **A**.

Consideremos ahora el caso de construir la traspuesta de una matriz bajo dicha representación.

Construir la traspuesta de una matriz consiste en mover sus datos de tal forma que si un elemento se halla en la fila **i**, columna **j**, en la traspuesta se hallará en la fila j, columna **i**.

Para la forma tradicional de representación de matrices, el algoritmo para hallar la traspuesta ***matb*** de una matriz ***mata*** es:

```
for i = 1  to  m  do
    for  j = 1  to  n  do
        matb(j,i) = mata(i,j)
    end(for)
end(for)
```

Y su orden de magnitud es **O(m*n)**, siendo **m** el número de filas y **n** el número de columnas de la matriz.

Para nuestra forma de representación en tripletas un primer algoritmo para hallar la traspuesta **B** de una matriz de tripletas **A** que representa una matriz ***mata*** es:

```
sub_programa traspuesta(A, B)
      t = A(1,3) + 1
      for  i = 1  to  t  do
           B(i,1) = A(i,2)
           B(i,2) = A(i,1)
           B(i,3) = A(i,3)
       end(for)
       ordene(B)
fin(traspuesta)
```

El ciclo FOR del procedimiento anterior, crea la traspuesta de la matriz representada en **A**, construyendo su representación en la matriz de tripletas **B**.

Sin embargo, la matriz de tripletas B creada no cumple las condiciones de orden definidas, lo cual implica un posterior proceso de ordenamiento. El orden de magnitud del procedimiento anterior es: para el ciclo FOR es **O(t)** y para el proceso de ordenamiento es $\mathbf{O(t^2)}$, siendo **t** el número de elementos diferentes de cero de la matriz ***mata***.

El caso más desfavorable se presenta cuando **t = n*m**, o sea, no hay elementos iguales a cero y el orden de magnitud de dicho algoritmo es **$O(n^2*m^2)$**, el cual es considerablemente mayor que para la representación tradicional: **O(n*m)**.

Del análisis anterior podemos concluir que por ahorrar espacio, estamos desmejorando notoriamente el tiempo de ejecución de una operación en particular.

La parte del algoritmo que hace que el tiempo sea malo, es el proceso de ordenamiento, si podemos evitar este proceso, lograremos mejorar el orden de magnitud.

Consideremos el siguiente algoritmo:

```
sub_programa traspuesta(A, B)
      B(1,1) = A(1,2)
      B(1,2) = A(1,1)
      B(1,3) = A(1,3)
      t = A(1,3) + 1
      k = 1
      n = A(1,2)
      for i = 1 to n do
            for j = 2 to t do
                  if A(j,2) = i then
                        k = k+1
                        B(k,1) = A(j,2)
                        B(k,2) = A(j,1)
                        B(k,3) = A(j,3)
                  end(if)
            end(for)
      end(for)
fin(traspuesta)
```

En el subprograma anterior se utiliza un ciclo externo con **i** desde 1, hasta **n**, o sea el número de columnas de la matriz ***mata***, para recorrer la matriz **A** buscando elementos cuya columna sea **i**, cuando se encuentra uno, se traslada a formar parte de la matriz traspuesta **B**.

De esta forma los elementos se pasan de acuerdo al orden establecido y nos evitamos el proceso de ordenamiento.

Establezcamos el orden de magnitud de dicho algoritmo: la instrucción IF se ejecuta **n*(t+1)** veces, lo cual implica un orden de magnitud **O(n*t)**. El caso más

desfavorable se presenta cuando **t = n*m** (todos los elementos son diferentes de cero) y el orden de magnitud es entonces **O(n²*m)** que comparado con **O(n*m)** de la forma tradicional es todavía malo.

Lo que hace inferior este algoritmo es que hay que recorrer toda la matriz **A** por cada columna que tenga la matriz ***mata*** que se está representando. Si logramos pasar todos los elementos de **A** recorriéndola sólo una vez, tendremos un algoritmo con orden de magnitud **O(t)**.

Para lograr lo anterior, lo que se necesita es conocer la posición en **B** a la cual hay que trasladar cualquier tripleta de **A** que se acceda. Para conocer dicha posición, debemos contar cuántos elementos hay en cada columna de la matriz que se está representando y determinar, de acuerdo a este número, las posiciones que le corresponden en la matriz de tripletas **B**.

El algoritmo para desarrollar esta labor lo presentamos a continuación:

```
sub_programa traspuesta(A, B)
      B(1,1) = A(1,2)
      B(1,2) = A(1,1)
      B(1,3) = A(1,3)
      n = A(1,2)
      t = A(1,3) + 1
      for i = 1 to n do
            s(i) = 0
      end(for)
      for i = 2 to t do
            s(A(i,2)) = s(A(i,2)) + 1
      end(for)
      p(1) = 2
      for i = 2 to n do
            p(i) = p(i-1) + s(i-1)
      end(for)
      for i = 2 to t do
            j = A(i,2)
            B(p(j),1) = A(i,2)
            B(p(j),2) = A(i,1)
            B(p(j),3) = A(i,3)
            p(j) = p(j) + 1
      end(for)
fin(traspuesta)
```

En el anterior subprograma se utiliza un vector **S**, el cual tiene tantos elementos como columnas tenga la matriz ***mata*** que se está representando. En el primer ciclo FOR se inicializan todos sus elementos en cero, en el segundo ciclo FOR se cuentan los elementos que hay en cada columna de ***mata***. Al terminar este ciclo si **S(3)** vale, digamos 4, significa que en la columna 3 de ***mata*** hay 4 elementos diferentes de cero. En general para un elemento **i** cualquiera de **S**, **S(i)** es el número de elementos diferentes de cero en la columna **i** de *mata*.

Se utiliza un vector **P** el cual contiene las posiciones de **B** a los cuales hay que trasladar una tripleta de **A**.

Inicialmente **P(1)** vale 2, indicando que la primera tripleta de la columna 1 de **A** será la segunda tripleta en la traspuesta **B**. En general, **P(i)=j**, indica que alguna tripleta de la columna **i** de la matriz ***mata***, será la j-èsima tripleta en la matriz de tripletas **B**.

El segundo y cuarto ciclo contribuyen en **O(t)** al orden de magnitud total del algoritmo. El caso más desfavorable, cuando **t** sea **n*m**, tendremos un algoritmo **O(n*m)** que es lo mismo que para la forma tradicional de representación.

Consideremos ahora el caso de multiplicación de matrices cuando se hallan representadas como matrices de tripletas.

A continuación presentamos un algoritmo mulmat(A,B,C), el cual multiplica las matrices A y B y deja el resultado en C.

```
sub_programa mulmat(A,B,C)
      m = A(1,1)
      n = A(1,2)
      na = A(1,3) + 1
      if  n <> B(1,1)  then
            write('no se puede multiplicar A con B')
            exit
      end(if)
      p = B(1,2)
      nb = B(1,3) + 1
      A(na+1,1) = n + 1
      traspuesta(B, BT)
      BT(nb+1,1) = p + 1
      BT(nb+1,2) = 0
      i = 2
      fila_actual = A(i,1)
```

```
inicio_fila_actual = i
k = 1
suma = 0
while i ≤ na do
        j = 2
        col_actual = BT(j,1)
        while  j ≤ nb + 1  do
                case
                        :A(i,1) <> fila_actual:
                                if  suma <> 0  then
                                        k = k + 1
                                        C(k,1) = fila_actual
                                        C(k,2) = col_actual
                                        C(k,3) = suma
                                        suma = 0
                                end(if)
                                while  BT(j,1) = col_actual  do
                                        j = j + 1
                                end(while)
                                col-actual = BT(j,1)
                                i = inicio_fila_actual
                        :BT(j,1) <> col_actual:
                                if  suma <> 0  then
                                        k = k + 1
                                        C(k,1) = fila_actual
                                        C(k,2) = col_actual
                                        C(k,3) = suma
                                        suma = 0
                                end(if)
                                col_actual = BT(j,1)
                                i = inicio_fila_actual
                        :A(i,2) < BT(j,2):
                                i = i + 1
                        :A(i,2) = BT(j,2):
                                suma = suma + A(i,3) * BT(j,3)
                                i = i + 1
                                j = j + 1
                        else
                                j = j + 1
                end(case)
        end(while)
        while A(i,1) = fila_actual do
                i = i + 1
        end(while)
```

```
            inicio_fila_actual = i
            fila_actual = A(i,1)
        end(while)
        C(1,1) = m
        C(1,2) = p
        C(1,3) = k - 1
fin(mulmat)
```

Los elementos de **C** se van almacenando en su lugar apropiado sin tener que mover resultados previamente calculados. Para lograr esto fijamos una columna de **A** y buscamos todos los elementos de una columna **j** de **B**. Obviamente, para encontrar los elementos de una columna en particular, debemos recorrer toda la matriz **B**. Para evitar esto calculamos primero la traspuesta de **B** la cual agrupa los elementos de todas las columnas.

Una vez localizados los elementos de una fila **i** de **A** y una columna **j** de **B** basta hacer un proceso de intercalación semejante al de suma de polinomios.

Analicemos el orden de magnitud: Las instrucciones antes de los ciclos While contribuyen en **O(p+nb)** el cual es debido al proceso de hallar la traspuesta de **B**.

El ciclo While externo se ejecuta a lo sumo **n** veces, (una vez por cada fila de **A**). En cada iteración del ciclo While interno, se incrementa el valor de **i**, o el de **j** o ambos, o, **i** y **col_actual** son restaurados a sus valores iniciales.

El máximo incremento de **j** es **NB**.

Si **Dr** es el número de términos en la fila **Fila_actual**, el valor de **i** se puede incrementar a lo sumo **Dr** veces antes de que **i** pase a la siguiente fila de **A**. Cuando esto sucede, **i** es restaurado a **inicio_fila_actual**, y al mismo tiempo **col_actual** avanza a la siguiente columna de **BT**.

Por consiguiente la restauración de **i** se puede ejecutar a lo sumo **p** veces (Hay sólo **p** columnas en **B**). Luego el máximo número de incrementos de **i** es **p*Dr**.

El máximo número de iteraciones del ciclo interno es entonces **p + pxDr + nb**, lo que da un orden de magnitud **O(p*Dr + nb).**

6.3 REPRESENTACIÓN DE MATRICES DISPERSAS COMO LISTAS LIGADAS, FORMA 1

A continuación abordaremos el problema de representar una matriz dispersa como lista ligada:

- Por cada fila que tenga la matriz tendremos una lista simplemente ligada circular con registro cabeza.
- Por cada columna que tenga la matriz también tendremos una lista simplemente ligada circular con registro cabeza.
- El registro cabeza de la lista de la fila **i** será el mismo de la lista de la columna **i**.
- Todo registro pertenecerá simultáneamente a dos listas: la lista de la fila **i** y la lista de la columna **j**.
- Habrá tantos registros cabeza como mayor sea el número de filas o columnas de la matriz a representar.

Para representar matrices como listas ligadas definiremos un registro así:

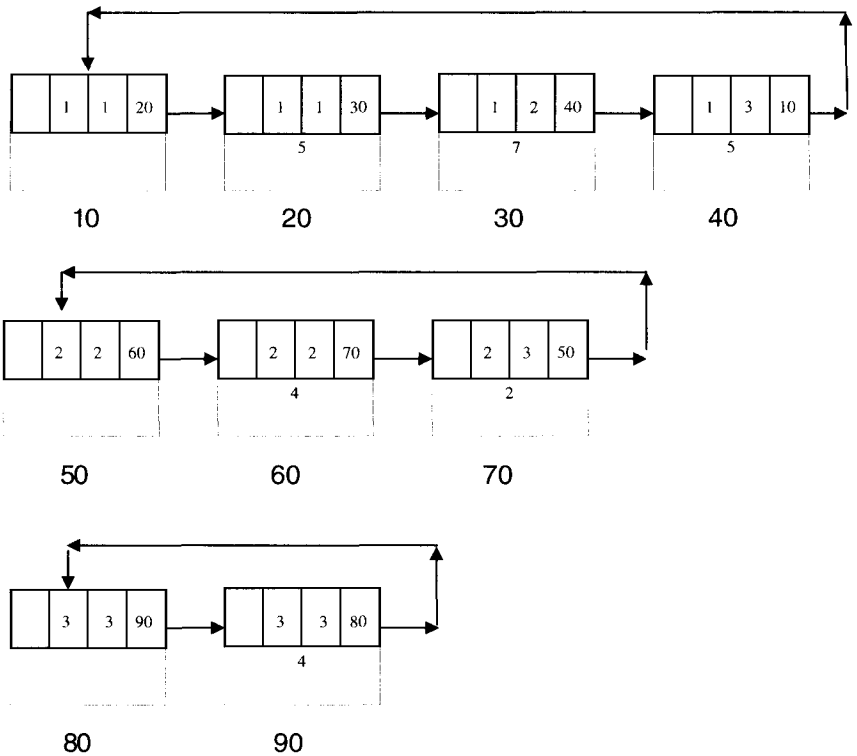
L.C	FILA	COLUMNA	L.F
VALOR			

El campo **LC** encadenará los registros por columnas. El campo **LF** encadenará los registros por filas. Los campos de **FILA**, **COLUMNA** y **VALOR** definirán el elemento diferente de cero que se está representando en ese nodo.

Consideremos la siguiente matriz:

mat	1	2	3
1	5	7	5
2		4	2
3			4

por cada fila tendremos una lista ligada como se muestra en la siguiente figura:



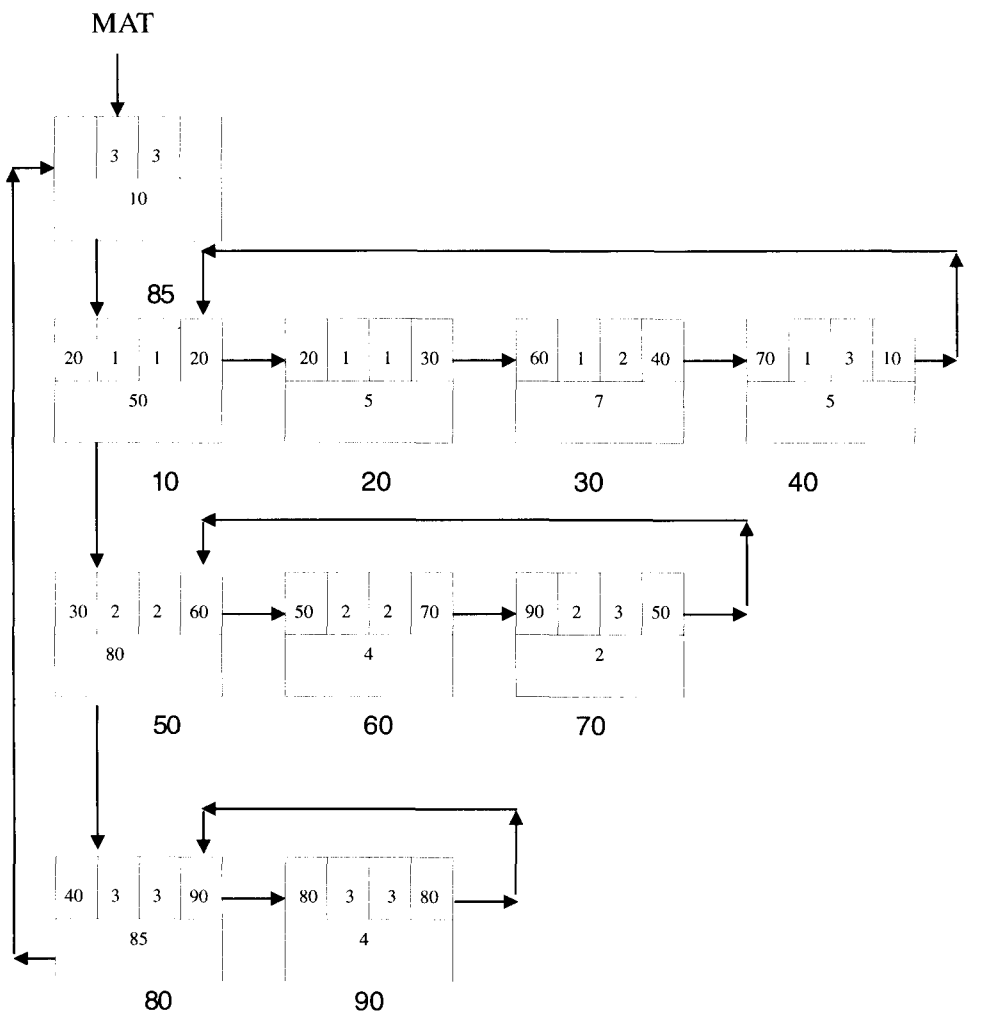
Para recorrer por columna encadenamos los mismos registros a través del campo **LC** de cada registro.

El campo de liga columna del registro 10, el cual es el registro cabeza de la lista que representa la fila 1, apuntará hacia el registro que represente el primer dato de la columna 1. En nuestro ejemplo es el mismo registro 20.

Como es inadecuado manejar un apuntador para cada registro cabeza, crearemos una lista simplemente ligada circular, con registro cabeza, con los registros cabeza de cada lista.

El apuntador para entrar a dicha lista lo llamaremos MAT.

La representación de la matriz del ejemplo será



El campo de liga columna del registro 50, el cual es el registro cabeza de las listas que representan la fila 2 y la columna 2, apuntará hacia el registro que represente el primer dato de la columna 2. En nuestro ejemplo el registro 30.

El campo de liga columna del registro 30 apuntará hacia el registro que represente el siguiente dato diferente de cero de la columna 2. En nuestro ejemplo el registro 60.

Veremos a continuación cómo crear la representación, como lista ligada, de una matriz dispersa. Consideremos la siguiente matriz:

Mat

	1	2	3	n = 4
1		4	8	5
2	2		7	
m = 3		1	5	

Los datos se entrarán por tripletas de la siguiente manera:

La primera tripleta contendrá las dimensiones de la matriz y el número de elementos diferentes de cero de la matriz. Las demás tripletas representarán, cada una, un elemento diferente de cero, ordenados ascendentemente por filas, y dentro de fila por columnas.

Para nuestro ejemplo, la entrada de datos es así:

```
m  n  r
3  4  7
f  c  v
1  2  4   Fila 1
1  3  8
1  4  5
-------
2  1  2   Fila 2
2  3  7
-------
3  2  1   Fila 3
3  3  5
```

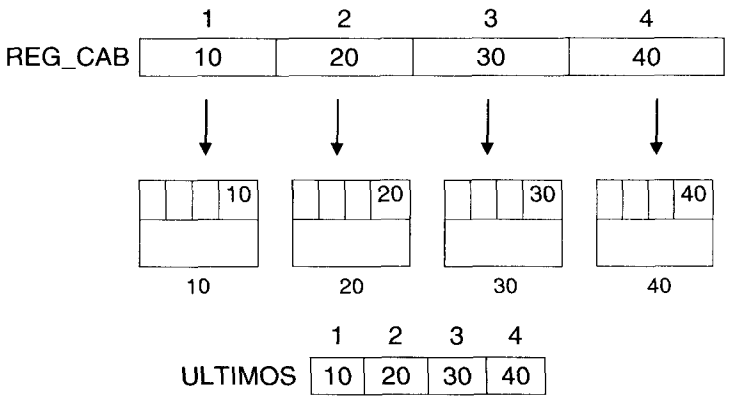
```
1     sub_programa leer_matriz(mat)
2           read(m,n,r)
3           p= mayor(m,n)
4           for  i = 1 to p  do
5                 conseguir_registro(x)
6                 If(x) = x
7                 reg_cab(i) = x
8                 ultimos(i) = x
9           end(for)
10          fila_actual = 1
11          ult = reg_cab(1)
12          for  i = 1 to r  do
13                read(f,c,v)
```

```
14                  if  f > fila_actual  then
15                        lf(ult) = reg_cab(fila_actual)
16                        fila_actual = f
17                        ult = reg_cab(f)
18                  end(if)
19                  conseguir_registro(x)
20                  fila(x) = f
21                  colu(x) = c
22                  valor(x) = v
23                  lf(ult) = x
24                  ult = x
25                  lc(ultimos(c)) = x
26                  ultimos(c) = x
27            end(for)
28            lf(ult) = reg_cab(fila_actual)
29            for i = 1 to p do
30                  lc(ultimos(i)) = reg_cab(i)
31            end (for)
32            for i = 1 to p -1 do
33                  valor(reg_cab(i)) = reg_cab(i+1)
34            end(for)
35            conseguir_registro(mat)
36            fila(mat) = m
37            colu(mat) = n
38            valor(mat) = reg_cab(1)
39            valor(reg_cab(p)) = mat
40      fin(leer_matriz).
```

- Instrucción 2 lee las dimensiones de la matriz y el número de elementos diferentes de cero.
- Instrucción 3 determina el mayor entre **m** y **n**, con el fin de conocer cuántos registros cabeza debemos conseguir.
- Instrucciones 4 a 9 consigue todos los registros cabeza y guarda la dirección de cada uno de ellos en el vector **reg_cab**.

El algoritmo construye simultáneamente la lista de una fila y las listas de todas las columnas. Como la forma de construirlas es añadiendo registros al final de ellas manejamos: una variable, llamada **ult**, para conservar el último de la lista de la fila que se está construyendo, y un vector, llamado **ultimos**, en el cual **ultimos(i)** mantiene la dirección del último registro de la lista de la columna **i**.

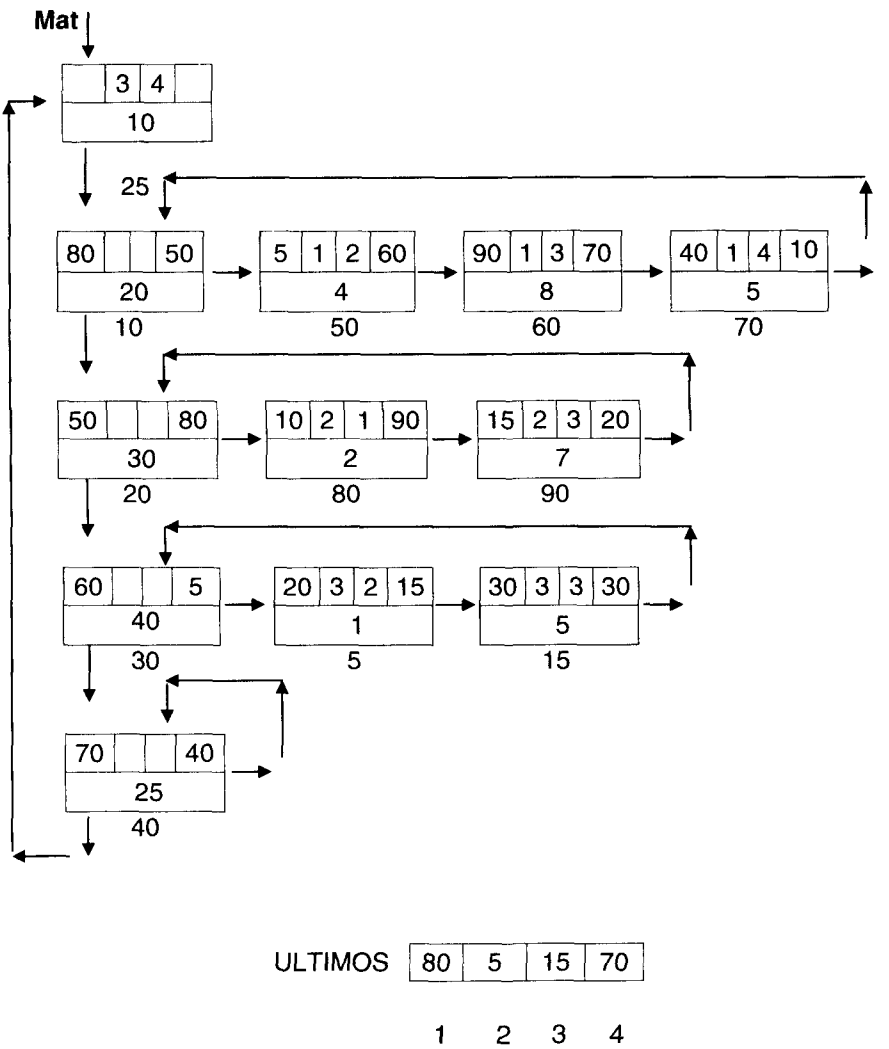
Terminadas de ejecutar las instrucciones 4 a 9 la situación es la siguiente:



La variable **fila_actual** mantendrá la información acerca de cuál es la fila que está actualmente en construcción.

- Instrucciones 10 y 11 asignan el valor de 1 a **fila_actual**, y **reg_cab(1)** a **ult,** puesto que es de suponer que en la fila 1 hay elementos diferentes de cero.
- Instrucciones 12 a 27 lee todos los datos de los elementos diferentes de cero.
- Instrucciones 14 a 18 actualizan **fila_actual, ult** y hace circular la lista de la fila que se terminó de construir.
- Instrucciones 19 a 22 consiguen nuevo registro y lo configuran con los datos de *fila, columna* y *valor leídos.*
- Instrucciones 23 y 24 encadenan dicho registro, a través del campo liga fila (LF), con el último de la lista de la fila que se está construyendo.
- Instrucciones 25 y 26 encadenan dicho registro, a través del campo liga columna (LC), con el último de lista de la columna **C**.
- Instrucción 28 hace circular la lista de la última fila que se construyó.
- Instrucciones 29 a 31 hacen circulares las listas de cada columna.
- Instrucciones 32 a 34 encadenan los registros cabeza a través del campo de **valor**.
- Instrucciones 35 a 39 consiguen el registro cabeza para la lista de registros cabeza y lo encadenan con el primero y el último de esta lista respectivamente.

Finalmente se tendrá:



Un algoritmo para recorrer una matriz representada como lista ligada de la anterior forma es:

```
sub_programa recorre_matriz(mat)
      p = valor(mat)
      while  p <> mat  do
            q = lf(p)
            while  q <> p  do
                  write(fila(q),col(q),valor(q))
                  q = lf(q)
```

```
            end(while)
            p = valor(p)
        edn(while)
end(sub_programa)
```

El algoritmo anterior, con el cual construimos la representación de matriz dispersa como forma 1 tiene la desventaja de que el usuario debe conocer cuántos elementos diferentes de cero tiene la matriz, además de tener que entrar los datos ordenados ascendentemente por filas y dentro de fila por columnas.

Presentamos a continuación otro algoritmo para construir la representación como lista ligada en forma 1, en la cual, se permitirá que el usuario entre los datos de cualquier forma, es decir, no tiene que entrar los datos ordenadamente ni saber cuántos elementos diferentes de cero tiene la matriz.

```
function construye_mat()
1.      read(m, n)
2.      mat = cons_reg_cab(m, n)
3.      mientras haya datos por leer haga
4.          read(f, c, v)
5.          new(x)
6.          fila(x) = f
7.          colu(x) = c
8.          valor(x) = v
9.          conecta_por_filas(mat, x)
10.         conecta_por_columnas(mat, x)
11.     fin(mientras)
12.     return(mat)
fin(construye_mat)
```

- Instrucción 1 lee las dimensiones de la matriz a representar.
- Instrucción 2 invoca la función **cons_reg_cab(m,n)**, la cual construye la lista simplemente ligada circular de registros cabeza, inicializando la lista de cada fila y de cada columna vacía.
- Instrucción 3 plantea el ciclo para lectura de los elementos diferentes de cero de la matriz.
- Instrucciones 5 a 8 consiguen un nuevo registro y los configura con los datos de fila columna y valor leídos en la instrucción 4.
- Instrucción 9 conecta el registro en la lista de la fila **f**, invocando el subprograma **conecta_por_filas**.
- Instrucción 10 conecta el registro en la lista de la columna **c**, invocando el subprograma **conecta_por_columnas**.

A continuación presentamos los subprogramas utilizados: construye registros cabeza y conecta por columnas.

```
function cons_reg_cab(m,n)
      may = mayor(m,n)
      new(x)
      mat = x
      fila(x) = m
      colu(x) = n
      ultimo = x
      for  i = 1  to  may  do
            new(x)
            fila(x) = x
            colu(x) = x
            Lf(x) = x
            Lc(x) = x
            valor(ultimo) = x
            ultimo = x
       end(for)
       valor(ultimo) = mat
       return(mat)
fin(cons_reg_cab)

sub_programa conecta_por_columnas(mat, x)
      p = mat
      for  i = 1  to  colu(x)  do
            p = valor(p)
       end(for)
       q = Lc(p)
       while  q <> p  and  fila(q) < fila(x)  do
            p = q
            q = Lc(q)
       end(while)
       Lc(x) = q
       Lc(p) = x
fin(conecta_por_columnas)

subprograma conecta_por_filas(mat, x)
      p = mat
      for  i = 1  to  fila(x)  do
            p = valor(p)
       end(for)
       q = Lf(p)
       while  q <> p  and  colu(q) < colu(x)  do
            p = q
```

```
            q = Lf(q)
        end(while)
        Lf(x) = q
        Lf(p) = x
fin(conecta_por_filas)
```

Por último presentamos un algoritmo para multiplicar dos matrices dispersas representadas como lista ligada forma 1.

```
Function mult_mat(A, B)
    if colu(A) <> fila(B) then
        write('matrices no multiplicables')
        return
    end(if)
    C = cons_reg_cab(fila(A), colu(B))
    p = valor(A)
    while p <> A do
        q = valor(B)
        while q <> B do
            s = 0
            r = Lf(p)
            t = Lc(p)
            while r <> p and t <> q do
                casos
                    :colu(r) < fila(t):
                        r = Lf(r)
                    :colu(r) > fila(t)
                        t = Lc(t)
                    :colu(r) = fila(t)
                        s = s + valor(r) * valor(t)
                        r = Lf(r)
                        t = Lc(t)
                fin(casos)
            end(while)
            if s <> 0 then
                new(x)
                fila(x) = fila(p)
                colu(x) = colu(q)
                conecta_por_filas(C, x)
                conecta_por_columnas(C, x)
            end(if)
            q = valor(q)
        end(while)
        p = valor(p)
    end(while)
```

```
    return(C)
fin(mult_mat)
```

6.4 REPRESENTACIÓN DE MATRICES DISPERSAS COMO LISTAS LIGADAS, FORMA 2

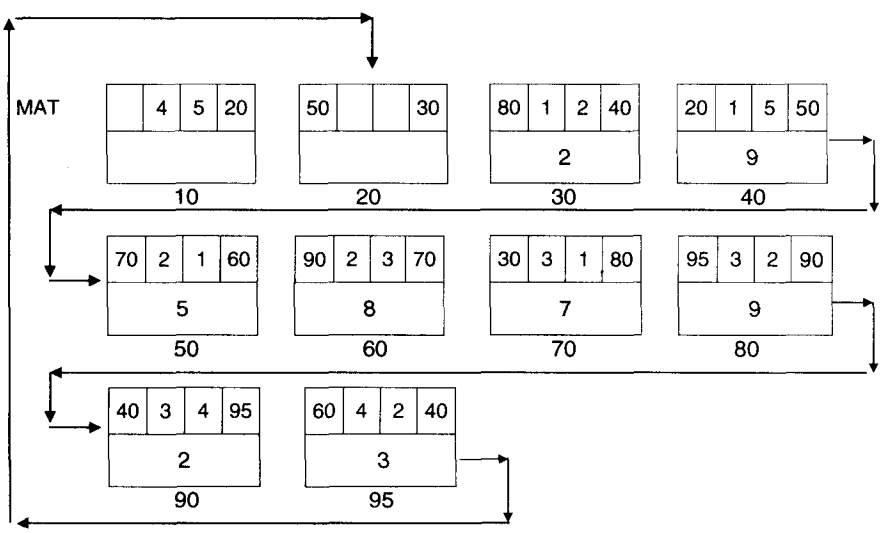
Consideraremos aquí una forma alterna para representar una matriz dispersa como lista ligada. Tomemos como ejemplo la siguiente matriz.

	1	2	3	4	5
1		2			9
2	5		8		
3	7	9		2	
4		3			

La configuración del registro sigue siendo la misma de la primera forma.

Con esta representación se tendrá un registro de entrada a la lista, (MAT), el cual, apunta hacia un registro auxiliar que será registro cabeza de dos listas ligadas: una con los registros encadenados ascendentemente por filas y dentro de fila por columnas, y otra con los registros encadenados por columnas y dentro de columna por filas.

La representación de la matriz anterior en la forma 2 se presenta en la figura.



6.5 REPRESENTACIÓN DE ARREGLOS EN MEMORIA

Aunque arreglos de varias dimensiones son suministrados como un objeto estandar de datos en la mayoría de los lenguajes de programación de alto nivel, es interesante ver cómo se representan en memoria. Recordemos que la memoria puede ser vista como un arreglo de registros de una dimensión, con ellos numerados desde 1 hasta **m**.

Por consiguiente, estamos enfrentados a representar arreglos de **n** dimensiones en una estructura de una dimensión. Aunque varias representaciones son factibles, debemos seleccionar una en la que la localización de un elemento del arreglo en memoria se pueda determinar eficientemente. Esto es necesario debido a que los programas que utilizan arreglos, por lo general, manejan los elementos del arreglo en forma aleatoria. Además de poder acceder los elementos fácilmente es también necesario determinar la cantidad de memoria que se debe reservar para un arreglo en particular. Asumiendo que cada elemento del arreglo requiere una palabra de memoria, el número de palabras necesarias es el número de elementos del arreglo. Si un arreglo es definido: $A(l_1{:}u_1, l_2{:}u_2, ..., l_n{:}u_n)$ el número de elementos es:

$$\prod_{i=1}^{n} (u_i - l_i + 1)$$

Una de las formas más comunes de representar arreglos es por filas. Si tenemos la definición:

A (4:5, 2:4, 1:2, 3:4)

El número de elementos es : 2 x 3 x 2 x 2 = 24

Utilizando la representación por filas, los elementos serían almacenados así:

A(4,2,1,3), A(4,2,1,4), A(4,2,2,3), A(4,2,2,4), A(4,3,1,3), A(4,3,1,4),
A(4,3,2,3), A(4,3,2,4), A(4,4,1,3), A(4,4,1,4), A(4,4,2,3), A(4,4,2,4),
A(5,2,1,3), A(5,2,1,4), A(5,2,2,3), A(5,2,2,4), A(5,3,1,3), A(5,3,1,4),
A(5,3,2,3), A(5,3,2,4), A(5,4,1,3), A(5,4,1,4), A(5,4,2,3), A(5,4,2,4)

en forma lineal.

Como se puede observar el subíndice de la derecha varía más rápidamente. Si miramos los subíndices como números, vemos que ellos están en orden ascendente:

4213, 4214, 4223, 4224, ... , 5414, 5423, 5424

Un sinónimo para representación por filas es orden lexicográfico. Desde el punto de vista de los compiladores, el problema es cómo obtener desde el nombre $A(i_1, i_2,...,i_n)$ la ubicación correcta en memoria. Supongamos que A(4,2,1,3) está almacenada en la localización 100 de la memoria, entonces A(4,2,1,4) estará en la posición 101, A(4,2,2,3) en la posición 102 y así sucesivamente, hasta llegar a A(5,4,2,4), que estará en la posición 123.

En general, podemos desarrollar una fórmula de direccionamiento para obtener la posición de cualquier elemento. Esta fórmula se construye con base en la dirección inicial y las dimensiones y subíndices del arreglo definido. Para simplificar la discusión asumiremos que el límite inferior de cada dimensión **i** es 1.

Antes de obtener una fórmula para un arreglo de N dimensiones, consideremos los casos particulares de arreglos de 1, 2 y 3 dimensiones.

Arreglos de una dimensión

Comencemos con un arreglo de una dimensión definido $A(1:u_1)$, asumamos una palabra por elemento. Entonces, en memoria se puede representar secuencialmente así:

Elemento del arreglo:	A(1),	A(2),	A(3),	... ,	$A(u_1)$
Dirección:	α,	$\alpha+1$,	$\alpha+2$,	... ,	$\alpha+u_1-1$

Si **α** es la posición (Dirección) de memoria en la cual se halla almacenado A(1), entonces A(2) se almacenará en la posición $\alpha+1$, A(3) estará en la posición $\alpha+2$, y la posición en la cual se halla almacenado cualquier elemento A(i) es:

$\alpha+(i-1)$.

Escribiremos la fórmula así: **POS = α + i - 1**

Si consideramos **α = 1** la fórmula quedaría **POS = i** Esta fórmula es lo que llamaremos fórmula de direccionamiento. De aquí en adelante, por simplicidad, consideraremos **α = 1**.

Arreglos de dos dimensiones

Consideremos la siguiente matriz MAT.

MAT	1	2	3	4	5
1	3	2	6	7	9
2	5	5	8	4	1
3	7	9	3	2	2

Nos interesa representar los datos linealmente en memoria. En un lenguaje de alto nivel para referirnos a algún elemento de la matriz lo hacemos utilizando dos subíndices: uno que hace referencia a la fila y otro que hace referencia a la columna. En general escribimos **MAT(i,j)** para hacer referencia al elemento de la fila **i**, columna **j**. Nuestro interés es obtener una fórmula en la cual conocidas las dimensiones **m** y **n** de la matriz (**m** número de filas, **n** número de columnas) y los subíndices **i** y **j** determinar en cuál posición de la memoria se almacena dicho dato.

	FILA 1					FILA 2					FILA 3				
POS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
memoria	3	2	6	7	9	5	5	8	4	1	7	9	3	2	2
(i, j)	1,1	1,2	1,3	1,4	1,5	2,1	2,2	2,3	2,4	2,5	3,1	3,2	3,3	3,4	3,5

En la representación por filas que hemos planteado, el elemento de la fila 1 columna 1 se halla en la posición 1 de memoria, el de la fila 2 columna 4 se halla en la posición 9 de memoria.

Para determinar una fórmula que nos establezca esta correspondencia utilizamos el método de inducción matemática, es decir, planteamos casos particulares y con base en ellos debemos ser capaces de deducir una fórmula general:

Para los elementos de la fila 1: POS = j
Para los elementos de la fila 2: POS = (los de la fila 1) + j
POS = n + j
Para los elementos de la fila 3: POS = (los de filas 1 y 2) + j
POS = 2*n + j
Para los elementos de la fila 4: POS = (los de las filas 1, 2 y 3) + j
POS = 3*n + j
" " "
" " "

Para los elementos de la fila **i**: POS = (los de las i -1 filas anteriores) + j
POS = (i -1)*n + j

O sea que la fórmula de direccionamiemto para representar, por filas, una matriz de **m** filas y **n** columnas, en forma lineal es:

POS = (i-1)*n + j

El ejemplo anterior nos muestra cómo determinar fórmulas de direccionamiento para representar, linealmente, arreglos de más de una dimensión.

El proceso viceversa también es interesante trabajarlo: conocidos m, **n**, **pos** y la fórmula de direccionamiento determinar los subíndices **i** y **j** correspondientes.

Para realizar este proceso se pueden elaborar algoritmos desde orden de magnitud cuadrático hasta orden de magnitud constante. Un algoritmo con orden de magnitud cuadrático es:

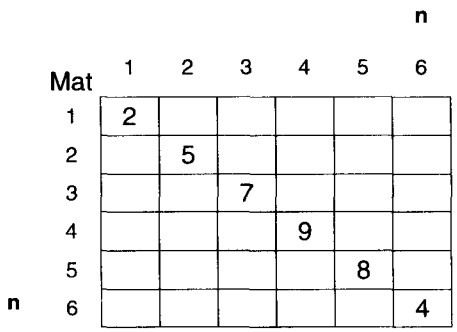
```
for i = 1 to m do
    for j = 1 to n do
        if pos = (i – 1)*n + j then
            return(i, j)
        end(if)
    end(for)
end(for)
```

y un algoritmo con orden de magnitud constante es:

```
i = pos / n
j = pos % n
if j = 0 then
   j = n
else
   i = i + 1
end(if)
```

De una manera análoga, hay cierto tipo de matrices dispersas, las cuales, con el objeto de ahorrar memoria, se pueden representar linealmente en vectores, para lo cual hay que hallar alguna fórmula de direccionamiento en el que conocidas las dimensiones de la matriz y los subíndices **i** y **j** de algún elemento, se pueda determinar la posición en el vector de dicho elemento. A esta forma de representación de matrices dispersas la denominaremos **por compresión**.

El caso más elemental se presenta con una matriz cuadrada en la cual sólo los elementos de la diagonal principal son diferentes de cero.



En nuestro ejemplo, **n**, el orden de la matriz, es 6.

Si representamos dicha matriz en la forma tradicional, consumimos $\mathbf{n^2}$ posiciones de memoria, de las cuales, sólo **n** estarán efectivamente siendo utilizadas, el resto de posiciones de memoria es desperdicio.

Para ahorrar memoria dicha matriz se puede representar en un vector utilizando fórmula de direccionamiento. La fórmula de direccionamiento para representarla es:

POS = i ó **POS = j**

y es válida sólo para aquellos elementos **mat(i,j)** que cumplan la condición i = j.

1	2	3	4	5	6
2	5	7	9	8	4

Es importante hacer énfasis en el hecho de que siempre que se deduzca una fórmula de direccionamiento hay que especificar para cuál rango de filas y columnas es válida dicha fórmula.

EJERCICIOS PROPUESTOS

1. Escriba un algoritmo que libere todos los registros de una matriz representada como lista ligada en la forma 1.

2. Escriba un algoritmo que sume dos matrices representadas como listas ligadas

3. Escriba un algoritmo que multiplique dos matrices representadas como listas ligadas.

4. Escriba un algoritmo que construya la traspuesta, en representación como lista ligada, de una matriz dispersa representada como lista ligada.

5. Elabore un algoritmo que determine el punto de silla de una matriz de M filas y N columnas. El punto de silla es el elemento A(i,j) tal que es el menor de la fila i y el mayor de la columna j.

6. Determine fórmula de direccionamiento para un arreglo de tres dimensiones.

7. Escriba algoritmos para determinar los valores de los subíndices i y j de un elemento de una matriz representada linealmente. Se conoce la posición, el orden de la matriz y la fórmula de direccionamiento. Su algoritmo puede ser lineal, cuadrático, logarítmico en base 2 o constante. Inténtelo de todas las formas.

8. Resuelva el punto 6 considerando la matriz representada en tripletas.

9. Elabore algoritmo para intercambiar dos filas de una matriz representada en tripletas.

10. Elabore algoritmo para intercambiar dos columnas en una matriz representada en tripletas.

11. Repita ejercicios 11 y 12 teniendo la matriz representada como lista ligada en ambas formas.

7

FÓRMULAS DE DIRECCIONAMIENTO EN MATRICES TRIANGULARES

Las matrices triangulares aparecen con frecuencia en la práctica, especialmente en la solución de sistemas de ecuaciones en el campo del Álgebra [1] y la Programación Lineal [2]. Dichas matrices son consideradas dispersas ya que muchos de sus elementos son cero. Así, por ejemplo, una matriz triangular de dimensión 50 x 50 tendrá al menos 1225 elementos en cero. Se prefiere representar estas matrices en vectores, llevando sus elementos ordenados ascendentemente por filas, y dentro de fila por columnas, a posiciones fijas en un vector, representando únicamente aquellos elementos diferentes de cero, logrando así un ahorro de memoria.

Dos problemas deben ser resueltos para lograr la correcta manipulación de dichas matrices. El primero: dado un elemento de la matriz, el cual se identifica por su fila y columna, debe obtenerse la posición que le corresponde en el vector y segundo, el problema «inverso»: dada una posición en el vector, determinar cuáles son las coordenadas del elemento de la matriz que allí se halla representado.

Las soluciones a estos dos problemas deben ser eficientes, pues de no ser así, se degradaría significativamente el tiempo de desempeño de los algoritmos que requiriesen tales transformaciones.

Se analizan a continuación las cuatro matrices triangulares: triangular inferior izquierda, triangular inferior derecha, triangular superior derecha y triangular superior izquierda, y para cada una de ellas se expone la forma de obtener las soluciones a los dos problemas mencionados.

7.1. MATRIZ TRIANGULAR INFERIOR IZQUIERDA (MTII)

En una matriz triangular inferior el número de elementos diferentes de cero será: uno en la fila uno, más dos en la fila dos, más tres en la fila tres y así sucesivamente, lo cual en términos matemáticos se obtiene mediante:

$$\sum_{i=1}^{i=n} i = n*(n+1)/2 \qquad \{f1\}$$

siendo **n** el orden de la matriz.

Esta fórmula es válida para cualquiera de las cuatro matrices triangulares aquí descritas.

	1	2	3	4	5	6	7
1	1						
2	2	3					
3	4	5	6				
4	7	8	9	10			
5	11	12	13	14	15		
6	16	17	18	19	20	21	
7	22	23	24	25	26	27	28

Figura 7.1. Matriz mtii

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

Figura 7. 2. Vector en el cual se representa la matriz mtii

Sea **mtii** matriz triangular inferior izquierda.

Sean **i** y **j** fila y columna de **mtii** y **pos** la posición del vector en la cual quedará almacenado el elemento de la fila **i** y columna **j** de **mtii**.

En todas las matrices aquí analizadas $\mathbf{1 \leq i \leq n}$ y $\mathbf{1 \leq j \leq n}$.

Para determinar una fórmula que permita conocer para un elemento de la fila **i** y columna **j**, utilizaremos el método de inducción matemática.

El análisis correspondiente se presenta en la tabla 7.1.

Tabla 7.1
Análisis para determinar pos en matriz triangular inferior izquierda

i	**j**	(los almacenados antes) +	?	**pos**
1	1	0	1	1
2	1	1	1	2
2	2	1	2	3
3	1	3	1	4
3	2	3	2	5
3	3	3	3	6
4	1	6	1	7
4	2	6	2	8
4	3	6	3	9
4	4	6	4	10
5	1	10	1	11
5	2	10	2	12
5	3	10	3	13
5	4	10	4	14
5	5	10	5	15
6	1	15	1	16
6	2	15	2	17
6	3	15	3	18
6	4	15	4	19
6	5	15	5	20
6	6	15	6	21
7	1	21	1	22
7	2	21	2	23
7	3	21	3	24
7	4	21	4	25
7	5	21	5	26
7	6	21	6	27
7	7	21	7	28

Para un elemento perteneciente a la fila **i** y columna **j** se observa que su posición **pos** en el vector se obtiene con base en la suma de los elementos de las **(i-1)** filas anteriores más la columna **j**, o sea:

$$pos = (\sum_{k=1}^{i-1} k) + j$$

Sumatoria cuyo valor es **i*(i-1)/2**, según **{f1}**

Por consiguiente: $pos = i*(i-1)/2 + j$ {f2}

El mismo resultado ha sido expuesto en [3] y [4].

Cuando se utilizan estas fórmulas se debe especificar para cuáles valores de **i** y **j** es válida la fórmula. En nuestro ejemplo la fórmula será válida sólo para aquellos elementos que pertenezcan a la triangular inferior, es decir:

$$\forall(i, j) / i \geq j$$

Una aplicación del uso de **{f2}** la podemos observar en el algoritmo de multiplicación de matrices, teniendo las matrices representadas en vectores de acuerdo a la figura 7.2 y dejando el resultado en un tercer vector que cumple la misma representación.

Tabla 7.2
Algoritmo que multiplica dos matrices triangulares inferiores representadas en vectores con f2.

```
// A, B: vectores de entrada que representan matrices a multiplicar.
// n: Orden de las matrices a multiplicar.
// C: vector de salida que representa la matriz resultado.

for  i = 1  to  n  do
     k = i*(i-1)/2
     for j = 1  to  i  do
          C(k+j)  =  0
          for  m = j  to  i  do
               C(k+j) = C(k+j) + A(k+m)*B(m*(m-1)/2+j)
          end(for)
     end(for)
end(for)
```

El Algoritmo de la tabla 7.2 tiene orden de magnitud cúbico, $O(n^3)$, el cual es un orden de magnitud típico para la multiplicación de matrices.

Ahora consideremos el proceso inverso, es decir, conocido **pos, n** y usando {f2} interesa determinar la fila y columna representadas en dicha posición.

Para ello se puede proceder de diversas formas.

I) Algoritmos cuadráticos. (2 formas).

Tabla 7.3
Algoritmo, con orden de magnitud cuadrático, para determinar fila y columna conocidos pos, n y {f2} en mtii. (versión 1)

```
// pos y n son parámetros de entrada
// fila y columna son parámetros de retorno

for k=1 to n
    for i=1 to k
       if (k*(k - 1) /2  + i) = pos
          fila = k
          columna = i
          return
       end(if)
    end(for)
end(for)
```

Tabla 7.4
Algoritmo, con orden de magnitud cuadrático, para determinar fila y columna conocidos pos, n y {f2} en mtii. (versión 2)

```
// pos y n son parámetros de entrada
// fila y columna parámetros de retorno

aux =1
for k=1 to n
    for i=1 to k
       if  aux  = pos  then
          fila = k
          columna = i
          return
       end(if)
       aux = aux +1
    end(for)
end(for)
```

En el peor de los casos, el orden de magnitud de los algoritmos de las tablas 7.3 y 7.4 es $\mathbf{O(n^2)}$, ya que cuando **pos** sea el último elemento de la matriz se tendrán que ejecutar los ciclos completamente.

II) Algoritmo lineal. Se propone una mejora del algoritmo anterior, la cual conduce a obtener un algoritmo con orden de magnitud lineal.

Tabla 7.5
Algoritmo, con orden de magnitud lineal, para determinar fila y columna conocidos pos, n y {f2} en una mtii.

```
// pos es parámetro de entrada
// fila y columna son parámetros de retorno

k=1
while  (k*(k + 1) /2 ) < pos  do
    k=k+1
end(while)
fila = k
columna = pos -  k*(k - 1) /2
```

El orden de magnitud del algoritmo de la tabla 7.5 es **O(n)** ya que en el peor de los casos, cuando **pos** está en la última fila, la **k** se incrementa **(n-1)** veces.

III) Algoritmo binario. Pero se puede mejorar aún más, haciendo uso de la idea de la búsqueda binaria [5] logrando un algoritmo con orden de magnitud $\mathbf{O(\log_2 n)}$. Dicho algoritmo se presenta en la tabla 7.6.

Tabla 7.6
Algoritmo, con orden de magnitud logarítmico, para determinar fila y columna conocidos pos, n y {f2} en mtii.

```
// pos y n son parámetros de entrada.
// fila y columna son parámetros de salida.

ki = 1
kf = n
while  ki <= kf   do
        km = ( ki + kf ) /2
        vf = km*(km +1) /2
        vi = vf – km
        casos
                : pos > vf :
                        ki = km +1
                : pos < = vi :
                        kf = km – 1
                : else :
                                fila = ki
                                columna =  pos –  fila*(fila – 1) /2
                                break
        fin(casos)
end(while)
```

IV) Algoritmo constante. Pero lo ideal es lograr obtener fila y columna sin realizar ciclos, obteniendo así un algoritmo con orden de magnitud **O(1)**. Se propone a continuación una manera de lograrlo.

De **{f2}** se tiene: **pos = i*(i -1)/2 + j**

La variable **j** en dicha fórmula se utiliza para determinar la posición **pos** en el vector de una fila dada, por consiguiente, si eliminamos esta variable de la fórmula de direccionamiento estaremos ubicados en la fila **i.**

La expresión **i*(i -1) /2** representa $\sum_{k=1}^{i-1} k$

Es decir, si tenemos la fórmula **pos = i*(i-1)/2**

Podremos despejar **i** y obtener la fila correspondiente a una posición **pos** dada. La variable **i** se despejará haciendo la siguiente transformación:

$2*pos = i*(i-1)$
$2*pos = i^2 - i$
$i^2 - i - 2*pos = 0$

la cual, es una expresión cuadrática cuya solución es:

i = (1+SQRT(1+8*pos))/2 **{f3}**

El análisis de los resultados obtenidos despejando **i** con la fórmula obtenida anteriormente se presenta en la tabla 7.7 columna 2.

Tabla 7.7
Análisis de resultados para determinar el valor de una fila conocidos pos, n y {f2} para una mtii.

pos	Valor de i (columna 2)	Valor de i (columna 3)
1	2	1
2	2.56	2
3	3	2.56
4	3.37	3
5	3.70	3.37
6	4	3.70
7	4.27	4
8	4.53	4.27
9	4.77	4.53
10	5	4.77
11	5.22	5
12	5.42	5.22
13	5.62	5.42
14	5.82	5.62
15	6	5.82
16	6.18	6
17	6.35	6.18
18	6.52	6.35
19	6.68	6.52
20	6.84	6.68
21	7	6.84
22	7.15	7
23	7.30	7.15
24	7.45	7.30
25	7.59	7.45
26	7.73	7.59
27	7.87	7.73
28	8	7.87

La **i** de la columna 3 se despejó de la fórmula que a continuación explicaremos: dada una posición **pos,** despejando la **i** y truncándola obtenemos la fila correspondiente a la posición **pos**, excepto para la diagonal principal, en la cual la **i** queda ubicada una fila más allá. Para corregir esto, restamos una unidad a **pos**, antes de despejar la **i**, o sea, despejamos **i** a partir de:

$$i*(i-1)/2 = pos-1 \qquad \{f3'\}$$

$$i = TRUNC((1+ SQRT(8*pos - 7) /2) \qquad \{f4\}$$

y la columna **j** se obtiene fácilmente de **{f2}**: **j = pos – i*(i - 1) /2**

Y nuestro algoritmo tendrá orden de magnitud **O(1)** y lo presentamos en la tabla 7.8.

Tabla 7.8
Algoritmo, con orden de magnitud constante, para determinar fila y columna, conocidos pos, n y {f2} para una mtii.

```
// pos parámetro de entrada
// fila y columna parámetros de retorno.

fila = TRUNC((1+ SQRT(8*pos - 7 ) /2)
columna = pos – fila*(fila - 1) /2
```

7.2. MATRIZ TRIANGULAR SUPERIOR DERECHA (MTSD).

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2		8	9	10	11	12	13
3			14	15	16	17	18
4				19	20	21	22
5					23	24	25
6						26	27
7							28

Figura 7.3. mtsd

Siguiendo las mismas convenciones tenemos:

Tabla 7.9
Análisis para determinar pos en mtsd.

i	j	(los almacenados antes) +	?	pos
1	1	0	1	1
1	2	0	2	2
1	3	0	3	3
1	4	0	4	4
1	5	0	5	5
1	6	0	6	6
1	7	0	7	7
2	2	7	1	8
2	3	7	2	9
2	4	7	3	10
2	5	7	4	11
2	6	7	5	12
2	7	7	6	13
3	3	13	1	14
3	4	13	2	15
3	5	13	3	16
3	6	13	4	17
3	7	13	5	18
4	4	18	1	19
4	5	18	2	20
4	6	18	3	21
4	7	18	4	22
5	5	22	1	23
5	6	22	2	24
5	7	22	3	25
6	6	25	1	26
6	7	25	2	27
7	7	27	1	28

Para un elemento situado en la fila **i** tenemos que sumar el número de elementos presentes en las **i – 1** filas anteriores así:

Si **i =5**

Se debe sumar 7+6+5+4

Esta sumatoria se expresa así: $\sum_{k=1}^{i-1} (n - k + 1)$

Ahora, se requiere determinar la componente **?** de la tabla 7.9 para determinar la posición **pos** en el vector.

Observando la tabla 7.9 tenemos:

Para la fila 2 columna 3 se debe sumar 2
Para la fila 3 columna 7 se debe sumar 5
Para la fila 4 columna 3 se debe sumar 2
Para la fila 5 columna 7 se debe sumar 3

El término **?** lo obtenemos mediante: **j - i +1**

O sea que: $$\mathbf{pos} = \sum_{k=1}^{i-1} (n - k + 1) + (j - i + 1)$$

la cual, desarrollando las sumatorias y factorizando se obtiene

$$\mathbf{pos} = (i - 1)(n - i/2) + j \qquad \{f6\}$$
$$\forall_{(i,j)} / j \geq i$$

Nótese que la división es real (No se trunca ni se redondea).

Un resultado y análisis similar puede ser visto en [7].

Para esta representación también se requiere desarrollar el proceso inverso, es decir, conocidos **pos**, **n** y la **{f6}** determinar la fila y columna correspondientes a esa posición **pos.** Se propone a continuación un algoritmo con orden de magnitud constante para lograrlo.

Partiendo de $$\mathbf{pos} = \sum_{k=1}^{i-1} (n - k + 1) + (j - i + 1)$$

El término que interesa para obtener la **fila** es el de la sumatoria, ya que el término **(j – i + 1)** sólo sirve para ubicar la posición **pos**, de una fila dada, en el vector.

Se tiene que: $$\sum_{k=1}^{i-1} (n - k + 1) = (i - 1)*(n - i/2 + 1)$$

Por tanto $\mathbf{pos} = (i - 1)*(n - i/2 + 1)$ **{f7}**

Utilizando nuevamente la fórmula para resolver una ecuación cuadrática despejamos **i** de **{f7}** y para una **mtsd** de orden 7 obtenemos la tabla 7.10.

Tabla 7.10
Resultados para determinar el valor de una fila conocidos pos, n y {f6} para una mtsd

Pos	Valor de i (columna 2)	Valor de i (columna 3)
1	1.13	1
2	1.27	1.13
3	1.41	1.27
4	1.55	1.41
5	1.70	1.55
6	1.85	1.70
7	2	1.85
8	2.16	2
9	2.31	2.16
10	2.48	2.31
11	2.65	2.48
12	2.82	2.65
13	3	2.82
14	3.18	3
15	3.38	3.18
16	3.58	3.38
17	3.78	3.58
18	4	3.78
19	4.23	4
20	4.47	4.23
21	4.73	4.47
22	5	4.73
23	5.30	5
24	5.63	5.30
25	6	5.63
26	6.44	6
27	7	6.44
28	8	7

Al solucionar la ecuación cuadrática se obtiene que las dos soluciones de la ecuación dan positivas, por consiguiente, sólo se toma la menor, ya que ésta cumple **i ≤ n.**

Como se presenta un desfase en una fila para los elementos situados en la última columna, similarmente como en la **mtii**, despejamos **i** pero restamos una unidad a la posición **pos** para corregir el desfase.

La tercera columna de la tabla 7.10 se obtiene entonces despejando **i** de la fórmula

$$\mathbf{pos - 1 = (i - 1)(n - i/2 + 1)} \qquad \{f8\}$$

La forma cuadrática que se obtiene es: $\mathbf{i^2 - i*(3+2*n) + 2(pos + n) = 0}$

Sea $\mathbf{W = 3 + 2*n}$

entonces: $i = TRUNC((W - SQRT(W^2 - 8(pos+n)))/ 2)$

y $j = pos - (i - 1)*(n - i/2)$

El algoritmo completo se presenta en la tabla 7.11.

Tabla 7.11
Algoritmo, con orden de magnitud constante, para determinar fila y columna, conocidos pos, n y {f6} en una mtsd.

```
// pos y n son parámetros de entrada
// fila y columna son parámetros de retorno

                    W = 3 + 2*n
fila = TRUNC((W – SQRT(W² - 8(pos + n) ))/ 2)
columna = pos - (fila -1)*(n - fila /2)
```

Sin embargo, también es posible lograrlo estableciendo la simetría con la triangular inferior izquierda.

Para establecer la correspondencia veamos la matriz enumerada de la siguiente manera:

	1	2	3	4	5	6	7
1	28	27	26	25	24	23	22
2		21	20	19	18	17	16
3			15	14	13	12	11
4				10	9	8	7
5					6	5	4
6						3	2
7							1

Figura 7.4

El número de elementos diferentes de cero en una matriz triangular es:

$$s = n(n+1)/2 \text{ por } \{f1\}.$$

Dada una posición **pos** cualquiera perteneciente a **mtsd**, si la restamos de **s** y aplicamos la fórmula **{f4}** obtenemos el complemento a **n** de la fila que se desea hallar, por tanto, si este valor se lo restamos a **n** obtendremos la fila real a la que corresponde esa posición en la **mtsd**. Despejar la columna será como en el caso anterior (última instrucción de la tabla 7.12).

En la tabla 7.13 presentamos el algoritmo.

Tabla 7.12
Algoritmo, con orden de magnitud constante, para determinar fila y columna, conocidos pos, n y {f6} en una mtsd.

```
// pos y n son parámetros de entrada
// fila y columna son parámetros de retorno

   posi = n (n+1) /2  - (pos -1)
   x = TRUNC((1+ SQRT(8*posi -7 ) /2 )
   fila = n – x +1
   columna = pos - (fila -1)*(n - fila /2)
```

Interesa demostrar la equivalencia de las fórmulas utilizadas en los algoritmos de las tablas 7.11 y 7.12 para determinar la fila.

De la tabla 7.11 se tiene que

$$\textbf{fila} = \textbf{TRUNC}((3+2n - \sqrt{4n^2 - 8pos + 4n + 9}) / 2)$$

y de la tabla 7.12 se tiene que

$$\textbf{fila} = (n+1) - \textbf{TRUNC}((1+\sqrt{4n^2 - 8pos + 1 + 4n}) / 2)$$

Para establecer la equivalencia, se requiere entrar el término **(n+1)** dentro de la función TRUNC, para ello se debe entrar como **(n+2)** y restar una unidad al término **pos** dentro del radical. Como se podrá comprobar, ésta sigue siendo una fórmula válida para el problema (ver tabla 7.13).

$$\text{fila} = \text{TRUNC}((n+2) - (1+ \sqrt{4n^2 - 8(pos - 1) + 1 + 4n}) / 2) \qquad \{f9\}$$

$$\text{fila} = \text{TRUNC}((2(n+2) - (1+ \sqrt{4n^2 - 8pos + 8 + 1 + 4n})) / 2)$$

la cual, al simplificarla, se obtiene:

$$fila = TRUNC((2*n+3 - SQRT(4*n^2 - 8*pos +4*n+9) /2)$$

la cual, es la misma fórmula extraída de la tabla 7.11.

Como una comprobación adicional presentamos la tabla 7.13 en la cual despejamos la fila utilizando ambas fórmulas.

Tabla 7.13
Comprobación de validez de algoritmos en tablas 7.11 y 7.12.

Pos	Valor fila dado por fórmula de tabla 7.11	Valor fila dado por fórmula de tabla 7.12	Valor fila según **{f9}**
1	Trunc(1) =1	8 - Trunc(7.865) =1	Trunc(9-8) = Trunc(1) =1
2	Trunc(1.135)=1	8 - Trunc(7.728) =1	Trunc(9-7.865) =Trunc(1.135)=1
3	Trunc(1.271)=1	8 - Trunc(7.588) =1	Trunc(9-7.728) =Trunc(1.272)=1
4	Trunc(1.411)=1	8 - Trunc(7.446) =1	Trunc(9-7.588) =Trunc(1.412)=1
5	Trunc(1.553)=1	8 - Trunc(7.300) =1	Trunc(9-7.446) =Trunc(1.554)=1
6	Trunc(1.699)=1	8 - Trunc(7.152) =1	Trunc(9-7.300) =Trunc(1.7) =1
7	Trunc(1.847)=1	8 - Trunc(7)	Trunc(9-7.152) =Trunc(1.848)=1

Nótese la equivalencia entre las columnas 2 y 4 de la tabla.

7.3. MATRIZ TRIANGULAR SUPERIOR IZQUIERDA (MTSI)

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	8	9	10	11	12	13	
3	14	15	16	17	18		
4	19	20	21	22			
5	23	24	25				
6	26	27					
7	28						

Figura 7.5. mtsi

Consideremos la tabla 7.14 para determinar la fórmula de direccionamiento.

Tabla 7.14
Análisis para determinar pos en mtsi

i	**j**	(los almacenados antes) +	?	**pos**
1	1	0	1	1
1	2	0	2	2
1	3	0	3	3
1	4	0	4	4
1	5	0	5	5
1	6	0	6	6
1	7	0	7	7
2	1	7	1	8
2	2	7	2	9
2	3	7	3	10
2	4	7	4	11
2	5	7	5	12
2	6	7	6	13
3	1	13	1	14
3	2	13	2	15
3	3	13	3	16
3	4	13	4	17
3	5	13	5	18
4	1	18	1	19
4	2	18	2	20
4	3	18	3	21
4	4	18	4	22
5	1	22	1	23
5	2	22	2	24
5	3	22	3	25
6	1	25	1	26
6	2	25	2	27
7	1	27	1	28

Para un elemento perteneciente a la fila **i** y columna **j** se observa que su posición **pos** en el vector se obtiene con base en la suma de los elementos de las **(i-1)** filas anteriores más la columna **j.**

Lo anterior se puede expresar así:

$$pos = \sum_{k=1}^{i-1}(n - k + 1) + j$$

Resolviendo la sumatoria se obtiene:

$$pos = (i - 1)(n - i/2 + 1) + j \qquad \{f10\}$$
$$\forall\ (i,j)/\ (j \leq n - i + 1)$$

De nuevo la división es real (no se trunca ni se redondea)

7.4. MATRIZ TRIANGULAR INFERIOR DERECHA (MTID)

	1	2	3	4	5	6	7
1							1
2						2	3
3					4	5	6
4				7	8	9	10
5			11	12	13	14	15
6		16	17	18	19	20	21
7	22	23	24	25	26	27	28

Figura 7.6. mtid

Consideremos la tabla 7.15 para determinar la fórmula de direccionamiento.

Tabla 7.15
Análisis para determinar pos en mtid

i	j	(los almacenados antes) +	?	pos
1	7	0	1	1
2	6	1	1	2
2	7	1	2	3
3	5	3	1	4
3	6	3	2	5
3	7	3	3	6
4	4	6	1	7
4	5	6	2	8
4	6	6	3	9
4	7	6	4	10
5	3	10	1	11
5	4	10	2	12
5	5	10	3	13
5	6	10	4	14
5	7	10	5	15
6	2	15	1	16
6	3	15	2	17
6	4	15	3	18
6	5	15	4	19
6	6	15	5	20
6	7	15	6	21
7	1	21	1	22
7	2	21	2	23
7	3	21	3	24
7	4	21	4	25
7	5	21	5	26
7	6	21	6	27
7	7	21	7	28

Para un elemento situado en la fila **i** tenemos que sumar el número de elementos presentes en las **(i-1)** filas anteriores así: Si **i = 5** Se debe sumar 1+2+3+4

es decir: $\sum_{k=1}^{i-1} k$

Ahora, se requiere determinar la componente **?** de la tabla 7.15 para determinar la posición **pos** en el vector.

Observando la tabla 7.15 tenemos:

Para la fila 2 columna 7 se debe sumar 2
Para la fila 3 columna 5 se debe sumar 1
Para la fila 4 columna 6 se debe sumar 3
Para la fila 5 columna 7 se debe sumar 5

El término **?** lo obtenemos mediante: **i + j - n**
por lo tanto:

$$\mathbf{pos} = \sum_{k=1}^{i-1} (k) + (i + j - n)$$

la cual, al desarrollar la sumatoria y simplificar se obtiene:

$$\mathbf{pos = i*(i+1)/2 + j - n} \qquad \{f11\}$$
$$\forall\ \mathbf{(i,j)/\ (j \geq n - i + 1)}$$

El análisis del proceso inverso para las matrices **mtsi** y **mtid** es similar al de las dos primeras matrices:

- Se observa que hay una correspondencia entre las matrices **mtsi** y **mtsd**. La fórmula para la fila es la misma, sólo cambia la de la columna así:

  columna = pos - (fila - 1)*(n - fila /2 + 1) **{f12}**

- Para la matriz **mtid** se obtiene la fila con la misma fórmula que para la matriz **mtii**; y la columna es:

  columna = pos – fila*(fila +1) /2 + n **{f13}**

8

PILAS

8.1 DEFINICIÓN

Una pila es una lista ordenada en la cual todas las operaciones (inserción y borrado) se efectúan en un solo extremo llamado TOPE. Es una estructura LIFO (**L**ast **I**nput **F**irst **O**utput) que son las iniciales de las palabras en inglés último en entrar primero en salir, debido a que los datos almacenados en ella se retiran en orden inverso al que fueron entrados.

Un ejemplo de pilas en computadores se presenta en el proceso de llamadas a subprogramas y sus retornos.

Supongamos que tenemos un programa principal y 3 subprogramas así:

Programa principal	Subprog. P1	Subprog. P2	Subprog. P3
---	---	---	---
---	---	---	---
---	---	---	---
---	P2	P3	---
P1	S ---	T ---	---
R ---	---	---	---
---	---	---	---
---	---	---	---
fin(prog. Ppal)	fin(P1)	fin(P2)	fin(P3)

Cuando se ejecuta el programa principal, se hace una llamada al subprograma P1, es decir, ocurre una interrupción a la ejecución del programa principal.

Antes de iniciar la ejecución de este subprograma, se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del programa principal cuando termine de ejecutar el subprograma. Llamemos R esta dirección.

Cuando ejecuta el subprograma P1 existe una llamada al subprograma P2, hay una nueva interrupción, pero antes de ejecutar el subprograma P2 se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del subprograma P1, cuando termine de ejecutar el subprograma P2. Llamemos S esta dirección.

Hasta el momento hay guardados dos direcciones de retorno: R, S

Cuando ejecuta el subprograma P2 hay llamada a un subprograma P3, lo cual implica una nueva interrupción y por ende guardar una dirección de retorno al subprograma P2, la cual llamamos T.

Tenemos entonces tres direcciones guardadas así: R, S, T

Al terminar la ejecución del subprograma P3, retorna a continuar ejecutando en la última dirección que guardó, es decir, extrae la dirección T y regresa a continuar ejecutando el subprograma P2 en dicha instrucción. Los datos guardados ya son:

R, S

Al terminar el subprograma P2, extrae la última dirección que tiene guardada y en este caso S, y retorna a esta dirección a continuar la ejecución del subprograma P1.

En este momento los datos guardados son: R

Al terminar la ejecución del subprograma P1 retornará a la dirección que tiene guardada, o sea a R.

Obsérvese que los datos fueron procesados en orden inverso al que fueron almacenados, es decir, último en entrar primero en salir. Es esta forma de procesamiento la que define una estructura PILA.

Definamos ahora formalmente la estructura pila:

```
ESTRUCTURA PILA
  Funciones:
    Crear()→ pila
    Apilar(elemento, pila) → pila
    Desapilar(pila) → pila
```

Tope(pila) → elemento
Esvacia(pila) → lógico

Axiomas:
Para todo P ∈ pilas, i ∈ elementos

Esvacia(Crear) ::= Verdad
Esvacia(Apilar(i, P)) ::= Falso
Desapilar(Crear) ::= Crear
Desapilar(Apilar(i, P)) ::= P
Tope(Crear) ::= error
Tope(Apilar(i, P)) ::= i

Crear, es la función constante que crea la pila vacía.

Apilar, es la función que permite incluir un elemento en una pila si tenemos una pila con 3 elementos R, S, T la estructura sería:

Apilar(T, Apilar(S, Apilar(R, Crear)))

De acuerdo a la forma general Apilar(i, P), el elemento i es T y

P = Apilar(S,Apilar(R,Crear))

Desapilar extrae el último elemento de la pila y deja una nueva pila la cual queda con un elemento menos.

Tope da información sobre el último elemento que se halla en la pila sin extraerlo.

Esvacia chequea que haya elementos en la pila.

8.2 REPRESENTACIÓN DE PILAS

Hemos definido la estructura pila en abstracto, veamos ahora cómo representar una pila en un computador.

Una pila dentro de un computador la podremos representar de dos formas: en un vector o como lista ligada.

Representación de pilas en un vector

La forma más simple es utilizar un arreglo de una dimensión y una variable, que llamaremos **Tope**, que indique la posición del arreglo en la cual se halla el último elemento de la pila.

La función **Crear** sería simplemente definir un vector y una variable, que llamaremos **Tope**, la cual inicialmente tendrá el valor cero.

```
dimension pila(100)
tope = 0
```

La función **Esvacia** consiste simplemente en preguntar por el valor de **Tope**:

```
if tope = 0 then
    verdad
else
    falso
end(if)
```

La función **Tope**, que da información sobre el último elemento de la pila es:

```
if tope = 0 then
   error
else
    return(pila(tope))
end(if)
```

Como se puede observar, estas funciones son tan sencillas y tan cortas que no se justifica elaborar algoritmos independientes para ellas. En su defecto se codificarán estas instrucciones, en el proceso que se esté efectuando, cuando sea necesario.

Los algoritmos para apilar y desapilar se presentan en los siguientes subprogramas:

```
sub_programa apilar(pila, n, tope, dato)
   if  tope = n then
       pila_llena
   else
      tope = tope + 1
      pila(tope) = dato
    end(if)
fin(apilar)
```

n y dato son parámetros por valor y **tope** y **pila** parámetros por referencia.

pila es un vector con capacidad de **n** elementos.

tope es la variable que indica la posición del vector en la cual se halla el último elemento de la pila.

dato es la variable que lleva la información a guardar en la pila.

Ejemplo:

	1	2	3	4	5	6	7	8	9
Pila	A	B	C	D					

En nuestro ejemplo el vector se llama **Pila**, **n** es 9 y **tope** es 4.
El subprograma PILA_LLENA se invocará cuando **tope** sea 9.

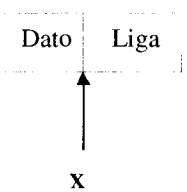
Dicho subprograma sacará un mensaje apropiado y detendrá el proceso.

```
sub_programa desapilar(pila,tope,dato)
   if  tope = 0  then
       pila_vacia
   else
       dato = pila(tope)
       tope = tope - 1
   end(if)
end(sub_programa)
```

pila, tope y **dato** son parámetros por variable.
dato es la variable que retorna la información sacada de la pila.

Representación de pilas como lista ligada

Siempre que se desee representar un objeto como lista ligada lo primero que se debe hacer es definir la configuración del registro.



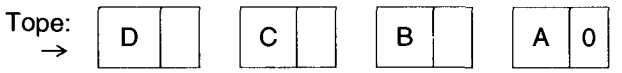
Dos campos: Dato y Liga

dato(x) = Se refiere al campo dato en el registro **x**.
liga(x) = Se refiere al campo liga del registro **x**, es el campo que apunta hacia otro registro.

Representemos como lista ligada la siguiente pila:

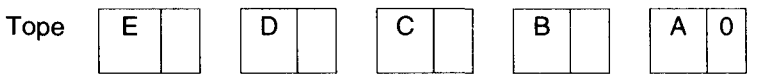
Pila

Dibujémosla de la forma como es común hacerlo con listas ligadas



Las operaciones son *Apilar y Desapilar.*

Apilar: Consiste en insertar un registro al principio de una lista ligada.




```
sub_programa apilar(tope,d)
      new(x)
      dato(x) = d
      liga(x) = tope
      tope = x
fin(apilar)
```

Se observa que es más fácil y más eficiente que manejando la pila en vectores, ya que en listas ligadas no hay necesidad de controlar pila llena antes de apilar.

Desapilar: consiste en borrar el primer registro de una lista ligada. Para desapilar es necesario saber si la pila está o no vacía. Si está vacía no se podrá desapilar.

La pila está vacía cuando la lista sea vacía, es decir, cuando **tope = 0**

```
sub_programa desapilar(tope, d)
    if  tope = 0 then
        pila_vacía
    else
        d = dato(tope)
        x = tope
        tope = liga(tope)
        devolver_registro(x)
fin(desapilar)
```

8.3 APLICACIÓN DE PILAS: MANEJO DE EXPRESIONES

Cuando los pioneros de la ciencia de los computadores concibieron la idea de lenguajes de programación de alto nivel, se enfrentaron a muchas dificultades técnicas. Una de las mayores fue cómo generar instrucciones en lenguaje de máquina que evaluaran correctamente una expresión aritmética.

Una asignación completa tal como: **X = A / B^C + D * E - A * C**

Puede tener varios significados; más aún si estuviera definida únicamente, es decir, utilizando paréntesis para definir el orden de ejecución de las operaciones, aún se ve dificultoso generar una secuencia correcta y razonable de instrucciones para evaluar dicha expresión. Afortunadamente la solución de que se dispone en la actualidad es simple y elegante. Más aún, es tan simple que este aspecto, en la elaboración de compiladores, es de los que requiere menor esfuerzo.

Una expresión está compuesta por operandos, operadores y delimitadores. La expresión anterior tiene 5 operandos : A, B, C, D y E.

Aunque en el ejemplo, las variables son de una letra, los operandos pueden ser cualquier nombre de variable o constantes en algún lenguaje de programación.

En cualquier lenguaje de programación los valores de las variables deben ser consistentes con las operaciones que se ejecutan sobre ellas. Estas operaciones son descritas por los operadores. En la mayoría de los lenguajes de programación hay diferentes operadores, los cuales se aplican a los diferentes tipos de datos que las variables pueden almacenar.

Primero están los operadores aritméticos básicos: Suma, resta, multiplicación, división y potenciación (+, -, *, /, ^). Otros operadores aritméticos son el más y el menos unitario.

Luego están los operadores relacionales: <, <=, =, >, >=, <>, los cuales generalmente se aplican sobre operadores aritméticos, aunque también se aplican a hileras de caracteres. ('GATO' es menor que 'PERRO' debido a su precedencia en orden alfabético). El resultado de una expresión que contiene operadores relacionales es o verdadero o falso.

Existen también los operadores lógicos, los cuales se aplican sobre expresiones o variables lógicas.

El primer problema para evaluar una expresión es decidir el orden en que se ejecutan las operaciones. Esto implica definir ese orden. Por ejemplo, si A = 4, B = 2, C = 2, D = 3 y E = 3 el valor asignado a X en la expresión dada podría ser:

```
X = 4/(2^2) + (3*3) - (4*2)
X = (4/4) + 9 - 8
X = 2
```

Sin embargo, la verdadera intención del programador podría haber sido:

X = (4/2)^(2 + 3) * (3-4) * 2
X = (4/2)^(5) * (-1) *2
X = (2^5) * (-2)
X = 32 * (-2)
X = - 64

Obviamente, él pudo haber especificado este último orden de evaluación, utilizando paréntesis:

X = ((((A / B)^(C + D))*(E - A))*C)

Para definir el orden de evaluación se ha asignado a cada operador una prioridad. Entonces, los operadores con mayor prioridad se ejecutan primero, los operadores con igual prioridad se ejecutan en el orden en que aparezcan de izquierda a derecha. Una muestra de prioridades de operadores es:

Operador	Prioridad
^	6
*, /	5
+, -	4
<, <=, =, >, >=, <>	3
NOT	2
AND	1
OR	0

Cuando se tiene una expresión con dos operaciones de la misma prioridad adyacentes, es necesario reglamentar cual se ejecuta primero. En álgebra se considera A^B^C como A^(B^C) y esta es la norma que rige para la potenciación cuando no hay paréntesis; la potenciación se ejecuta de derecha a izquierda. Sin embargo, expresiones como A*B/C ó A+B-C se evalúan de izquierda a derecha.

Recuerde que este orden se puede alterar utilizando paréntesis, en cuyo caso la evaluación se efectúa desde los paréntesis más internos hacia los más externos.

Ahora que hemos definido prioridades sabemos cómo se evalúa correctamente la expresión dada.

Ahora, ¿cómo hace un compilador para evaluar una expresión dada?
La respuesta se obtiene reescribiendo la expresión en una forma llamada notación **posfijo**.

La forma tradicional como escribimos las expresiones se denomina **infijo**, porque los operadores van precedidos y seguidos por operandos.

Realmente existen 3 formas de escribir expresiones. Si queremos indicar que hay que sumar **a** con **b** lo podremos hacer así:

- Notación **INFIJO**: Operando operador operando **a+b**.
- Notación **POSFIJO**: Operando operando operador **ab+**.
- Notación **PREFIJO**: Operador operando operando **+ab**.

Cuando se tiene una expresión POSFIJO, el operador actúa sobre los dos operandos que le preceden, sin importar la prioridad de ejecución de los operadores y sin tener que utilizar paréntesis.

Cuando se tiene una expresión PREFIJO, el operador actúa sobre los dos operandos que le suceden, sin importar la prioridad de ejecución de los operadores y sin tener que utilizar paréntesis.

Consideremos la expresión que hemos estado trabajando: Su forma **POSFIJO** es: **ABC^/DE*+AC*-**

Hagamos un seguimiento a la forma de evaluación.

Cada vez que se efectúe una operación almacenamos su resultado en una posición temporal.

Procesando de izquierda a derecha, el primer operador que se encuentra es la potenciación, la cual se aplica sobre los dos operandos que le proceden que son B y C. A continuación damos una tabla del orden de ejecución de las operaciones y la forma como va quedando la expresión posfijo.

OPERACION		POSFIJO
R1 = B^C	→	A R1 / DE * + AC * -
R2 = A/R1	→	R2 DE * + AC * -
R3 = D * E	→	R2 R3 + AC * -
R4 = R2 + R3	→	R4 AC * -
R5 = A*C	→	R4 R5 -
R6 = R4 - R5	→	R6

Y R6 contiene el resultado.

Como se puede ver los paréntesis ya no son necesarios y la prioridad de los operadores ya no es relevante. La expresión se evalúa haciendo una búsqueda de izquierda a derecha, apilando operandos y aplicando los operadores sobre los dos últimos operandos que se hallen en la pila y colocando el resultado en la pila.

Este proceso de evaluación es mucho más simple que intentar una evaluación directa sobre la notación infijo.

Escribamos ahora un algoritmo para evaluar una expresión en notación POSFIJO.

```
sub_programa evapos(e)
    dimension pila(100)
    tope = 0
    x = siguiente_token(e)
    while  x <> ' '  do
        if x in operadores then
            opn2 = pila(tope)
            opn1 = pila(tope-1)
            casos de  x
                :*:  res = opn1 * opn2
                :/:  res = opn1 / opn2
                :+: res = opn1 + opn2
                :-:  res = opn1 - opn2
            fin(casos)
            tope = tope - 1
            pila(tope) = res
        else
            tope = tope + 1
            pila(tope) = x
        end(if)
        x = siguiente_token(e)
    end(while)
    res = pila(tope)
fin(evapos)
```

El anterior algoritmo utiliza una función llamada SIGUIENTE_TOKEN.

Cuando se tiene una expresión los operandos y operadores no necesariamente son de un carácter. En general, cualquier elemento de una expresión, operando u operador, recibe el nombre de TOKEN, la función SIGUIENTE_TOKEN, se encarga de extraer de la expresión, el elemento a procesar, sin importar el número de caracteres que tenga.

Veamos ahora cómo convertir una expresión en notación INFIJO a notación POSFIJO. Los pasos a seguir son:

- Parentizar completamente la expresión infijo.
- Mover los operadores a reemplazar su correspondiente paréntesis derecho.
- Borrar todos los paréntesis izquierdos.

Como ejemplo consideremos la expresión: A / B^C + D * E - A * C

Parentizarla completamente de acuerdo al orden de ejecución de los operadores. A cada operador le corresponderá un paréntesis izquierdo y un paréntesis derecho.

(((A / (B^C)) + (D * E))- (A * C))

Los operadores se moverán a reemplazar su correspondiente paréntesis derecho.

Al hacer los movimientos la expresión queda: (((A (BC^ / (DE * + (AC * -

Reescribir la expresión suprimiendo los paréntesis izquierdos. La expresión quedará:

ABC^ /DE * + AC * -

El problema con este método es que requiere varios pasos: parentizar la expresión, mover operadores y suprimir paréntesis izquierdos.

Si observamos bien ambas expresiones (en infijo y en posfijo) vemos que los operandos conservan el mismo orden. Por tanto si recorremos la expresión infijo de izquierda a derecha, podemos comenzar a armar la expresión posfijo, escribiendo los operandos a medida que se encuentren y el problema se reduce a un manejo de operadores, es decir, en qué momento se deben incluír en la expresión posfijo. La solución es almacenarlos en una pila y determinar el momento preciso en el cual se deben pasar a la expresión posfijo.

Por ejemplo, si tenemos la expresión A + B * C y deseamos llegar a su forma posfijo ABC*+ la secuencia de apilar y desapilar sería:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
A	Vacía	A
+	+	A
B	+	AB

En este punto el SIGUIENTE_TOKEN es el operador * y debemos determinar si se coloca en la pila o si desapilamos el operador +. Debido a que el * es de mayor prioridad que el + lo apilamos:

*	+ *	AB
C	+ *	ABC

Hemos terminado con la expresión infijo y lo que hay que hacer es sacar los operadores que tenemos en la pila y llevarlos a la expresión obteniendo ABC*+.

Consideremos otro ejemplo: A*(B+C)*D, su forma posfijo es ABC+*D*, veamos la secuencia de operaciones para obtener esta expresión:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
A	Vacía	A
*	*	A
(	* (	A
B	* (	AB
+	* (+	AB
C	* (+	ABC

En este momento el siguiente token es), la acción a tomar es desapilar todos los operadores hasta encontrar el paréntesis izquierdo correspondiente y luego sacarlo de la pila sin incluirlo en la expresión posfijo:

)	*	ABC +

El siguiente token es un asterisco, y el operador que está en la pila es también un asterisco; como son operadores de igual prioridad entonces sacamos el operador de la pila y lo llevamos a la expresión posfijo y apilamos el operador que se acabó de leer:

*	*	ABC + *
D	*	ABC + * D

Terminamos la expresión infijo, entonces sacamos los operadores que hay en la pila y obtenemos ABC + * D * que es la expresión posfija buscada.

En general, cuando se lea un operador siempre debe ser llevado a la pila; sin embargo, antes de hacer esto, debemos sacar de la pila todos los operadores cuya prioridad sea mayor o igual que la prioridad del operador que va a entrar.

Cuando se encuentre un paréntesis izquierdo siempre debe ser llevado a la pila sin sacar operadores de ella.

Cuando se encuentre un paréntesis derecho, se deben sacar todos los operadores que haya en la pila hacia la expresión posfijo, hasta hallar su correspondiente paréntesis izquierdo, el cual se borra de la pila sin llevarlo a la expresión posfijo.

Consideremos el caso de la potenciación, la cual se ejecuta de derecha a izquierda. Si tenemos la expresión A^B^C su forma posfijo es ABC^^

Al aplicar el proceso establecido tenemos:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
A	Vacía	A
^	^	A
B	^	A B

El siguiente token es otro operador de potenciación y como la prioridad del operador que está en la pila es igual a la del operador que va a entrar, la acción es extraer el operador de la pila e incluir el que viene, por tanto obtenemos:

^	^	A B ^
C	^	A B ^ C

Se terminó la expresión infijo, luego la expresión posfijo resultante será:

AB^C^

la cual es diferente a la expresión posfija correcta que es ABC^^

Lo anterior nos lleva a adoptar una condición diferente para los operadores de operaciones que se ejecutan de derecha a izquierda. La solución a este problema es asignar prioridad distinta de ejecución a estos operadores para cuando están dentro de la pila y para cuando están fuera de la pila. Como el operador dentro de la pila no puede salir, le asignamos a estos operadores una prioridad mayor fuera de la pila que dentro de la pila.

Todas estas condiciones anteriores nos llevan a construir una tabla de prioridad de los operadores dentro y fuera de la pila, así:

OPERADOR	PRIORIDAD DENTRO DE LA PILA	PRIORIDAD FUERA DE LA PILA
^	3	4
*	2	2
/	2	2
+	1	1
-	1	1
(	0	4

El subprograma para convertir una expresión INFIJO a POSFIJO es:

```
sub_programa intopos(e)
      dimension pila(100)
      tope = 0
      x = siguiente_token(e)
      while x <> ' ' do
          casos de x
             :x in operadores:
                  while tope > 0 and pdp(pila(tope)) >= pfp(x) do
                        write (pila(tope))
                        tope = tope - 1
                  end(while)
                  tope = tope + 1
                  pila(tope) = x
             :x = ')':
                  while pila(tope) <> '(' do
                        write(pila(tope))
                        tope = tope - 1
                  end(while)
                  tope = tope - 1
             else
                  write(x)
          fin(casos)
          x = siguiente-token(e)
      end(while)
      while tope > 0 do
          write(pila(tope))
          tope = tope - 1
      end(while)
fin(intopos)
```

EJERCICIOS PROPUESTOS

1. Elabore un algoritmo para traducir una expresión de infijo a prefijo.
2. Elabore un algoritmo para traducir una expresión de posfijo a prefijo.
3. Elabore un algortimo para traducir una expresión de posfijo a infijo.

4. Elabore un algoritmo para traducir una expresión de prefijo a infijo.

5. Elabore un algoritmo para traducir una expresión de prefijo a posfijo.

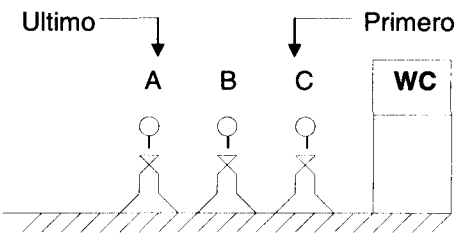
6. Elabore los algoritmos anteriores considerando las expresiones representadas como listas ligadas con todas sus posibilidades.

9

COLAS

9.1 DEFINICIÓN

Una cola es una lista ordenada en la cual las operaciones de inserción se efectúan en un extremo llamado **ultimo** y las operaciones de borrado se efectúan en el otro extremo llamado **primero**.



En términos prácticos es lo que supuestamente se debe hacer para tomar un bus, para comprar las boletas para entrar a un cine o para hacer uso de un servicio público.

Las operaciones sobre una cola serían entonces:

Crear() la cual crea una cola vacía.
Encolar(i, C) la cual añade un elemento i al final de la cola.
Desencolar(C) la cual remueve el primer elemento de la cola.
Primero(C) la cual da el primer elemento de la cola.
Esvacia(C) la cual es falsa o verdadera dependiendo de si la cola está vacía o no.

La especificación completa de esta estructura de datos es:

ESTRUCTURA COLA
Funciones:
Crear()→ cola
Esvacia(cola) → lógico
Encolar(elemento, cola) → cola
Desencolar(cola) → cola
Primero(cola) → elemento

Axiomas:
Para todo C ∈ colas, i ∈ elementos
Esvacia(crear) ::= Verdad
Esvacia(Encolar(i, C)) ::= Falso
Primero(Crear) ::= error
Primero(Encolar(i, C)) ::= if Esvacia(C) then i
else Primero(C)
Desencolar(Crear) ::= Crear
Desencolar(Encolar(i, C)) ::= if Esvacia(C) then Crear
else Encolar(i, Desencolar(C))

Si tenemos una cola: **a, b, c** donde **a** es el primer elemento de la cola y **c** es el último, de acuerdo a las funciones definidas la cola es así:

Encolar(**c**,Encolar(**b**,Encolar(**a**,Crear)))

Dejo al estudiante la tarea de probar si son correctos los axiomas definidos.

9.2 REPRESENTACION DE COLAS

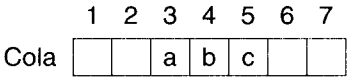
En general las colas se pueden representar en vector o como lista ligada. En la representación en vector se puede hacer de dos formas: en forma no circular y en forma circular. Veremos las tres formas de representación.

Representación de colas en un vector, en forma no circular

Para representar colas en esta forma se requiere un vector, que llamaremos **cola** y dos variables: una que llamaremos **primero**, que indique la posición del vector en la cual se halla el primer dato de la cola, y otra, que llamaremos **ultimo**, la cual indica la posición en la cual se halla el último dato de la cola.

En el vector se guardan los datos, **primero** apuntará hacia la posición anterior en la cual se halla realmente el primer dato de la cola, **ultimo** apuntará hacia la posición en la que realmente se halla el último dato de la cola.

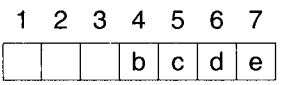
Si tenemos un vector con la siguiente configuración:



La variable **primero** valdrá 2 indicando que el primer elemento de la cola se halla en la posición 3 del vector; la variable **ultimo** valdrá 5 indicando que el último elemento de la cola se halla en la posición 5 del vector.

Las operaciones sobre la cola que son encolar y desencolar funcionan de la siguiente forma: si se desea encolar basta con incrementar la variable **ultimo** en uno y llevar a la posición **ultimo** del vector el dato a encolar; si se desea desencolar basta con incrementar en uno la variable **primero** y extraer de dicha posición el dato que allí se halla.

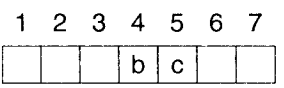
En general, si el vector **cola** definido tiene **n** elementos, cuando la variable **ultimo** sea **n**, se podría pensar que la cola está llena.



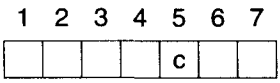
Pero, como se observa en la figura el vector tiene espacio al principio, por consiguiente podemos pensar e mover los datos hacia el extremo izquierdo, actualizar **primero** y **ultimo** para luego encolar.

En otras palabras, la condición de cola llena será cuando **primero** sea igual a cero y **ultimo** sea igual a **n**.

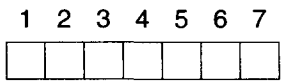
Consideremos ahora operaciones sucesivas de desencole para el ejemplo dado: después de ejecutar la primera operación de desencole los valores de las variables **ultimo** y **primero** y del vector **cola** será la siguiente:



donde la variable **primero** valdrá 3 y la variable **ultimo** 5.
Al desencolar nuevamente, la configuración del vector será:



donde la variable **primero** vale 4 y la variable **ultimo** 5.
Si desencolamos de nuevo, la configuración del vector será:



y la variable **primero** vale 5 y la variable **ultimo** 5.

En otras palabras **primero** es igual a **ultimo** y la cola está vacía, o sea, la condición de cola vacía será **primero = ultimo.**

Teniendo definidas las condiciones de cola llena y cola vacía podemos proceder a escribir los algoritmos para encolar y desencolar representando la cola en un vector de **N** elementos.

```
sub_programa encolar(cola, n, primero, ultimo, d)
      if  ultimo = n  then
            if primero = 0  then
                  cola_llena
            else
            for i = primero + 1  to  n  do
                  cola(i - primero) = cola(i)
            end(for)
            ultimo = ultimo - primero
            primero = 0
      end(if)
      ultimo = ultimo + 1
      cola(ultimo) = d
fin(encolar)
sub_programa desencolar(cola, primero, ultimo, d)
      if  primero = ultimo  then
            cola_vacia
      end(if)
      primero = primero + 1
      d = cola(primero)
fin(desencolar)
```

En el subprograma para encolar los parámetros **cola, n, primero, ultimo** y **d** son: el vector, el número de elementos de éste, la posición del primer elemento de la cola, la posición del último elemento de la cola y el dato a encolar, respectivamente.

En el subprograma desencolar **cola** es el vector, **primero** y **ultimo** son las variables para identificar las posiciones donde se hallan el primero y el último elemento de la cola en el vector, tal como se definió anteriormente y **d** es la variable donde regresa el dato desencolado.

si consideramos una situación como la siguiente:

1	2	3	4	5	6	7
	e	f	b	c	d	g

n vale 7, **ultimo** vale 7 y **primero** vale 1.

Si se desea encolar el dato **h** y aplicamos el subprograma ENCOLAR definido, la acción que efectúa dicho subprograma es mover los elementos desde la posición 2 hasta la 7, una posición hacia la izquierda de tal forma que quede espacio en el vector para incluir la **h** en la cola. El vector quedará así:

1	2	3	4	5	6	7
e	f	b	c	d	g	

con **primero** valiendo 0 y **ultimo** valiendo 6. De esta forma continúa la ejecución del subprograma **encolar** y se incluye el dato **h** en la posición 7 del vector. El vector queda así:

1	2	3	4	5	6	7
e	f	b	c	d	g	h

Con **primero** valiendo 0 y **ultimo** valiendo 7.
Si en este momento se desencola el vector queda así:

1	2	3	4	5	6	7
	f	b	c	d	g	h

con **primero** valiendo 1 y **ultimo** valiendo 7.

Si la situación anterior se repite sucesivas veces, el cual sería el peor de los casos, cada vez que se vaya a encolar se tendrían que mover **n-1** elementos en el vector, lo cual haría ineficiente el manejo de la cola, ya que el proceso de encolar tendría orden de magnitud **O(n)**.

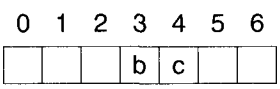
Para obviar este problema y poder efectuar los subprogramas de ENCOLAR y DESENCOLAR en un tiempo O(1) manejaremos el vector circularmente.

Representación de colas circularmente en un vector

Para manejar una cola circularmente en un vector se requiere definir un vector de **n** elementos con los subíndices en el rango desde 0 hasta **n-1**, es decir, si el vector tiene 10 elementos los subíndices variarán desde 0 hasta 9, y el incremento de las variables **primero** y **ultimo** se hace utilizando la función u operación **módulo** de la siguiente manera:

```
primero = (primero + 1) % n
ultimo = (ultimo + 1) % n
```

Dicha operación retorna el residuo de una división entera. Si se tiene una cola en un vector, con la siguiente configuración:

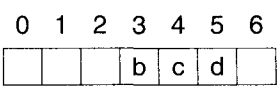


n vale 7, los subíndices varían desde 0 hasta 6, **primero** vale 2 y **ultimo** vale 4.

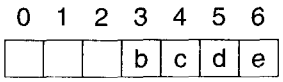
Si se desea encolar el dato **d** incrementamos la variable **ultimo** utilizando la operación **módulo** así:

ultimo = (ultimo + 1) % n

o sea, **ultimo = (4+1) % 7**, la cual asignará a **ultimo** el residuo de dividir 5 por 7, que es 5. El vector queda así:



Al encolar el dato **e** el vector queda así:

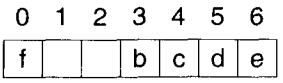


y las variables **primero** y **ultimo** valdrán 2 y 6 respectivamente.

Al encolar de nuevo, digamos el dato **f,** aplicamos el operador % obteniendo el siguiente resultado:

ultimo = (6+1) % 7, el cual es 0, ya que el residuo de dividir 7 por 7 es cero.

Por consiguiente la posición del vector a la cual se llevará el dato **f** es la posición 0. El vector quedará con la siguiente conformación:



con **ultimo** valiendo 0 y **primero** 2. De esta forma hemos podido encolar en el vector sin necesidad de mover elementos en él.

Los subprogramas para encolar y desencolar quedan así:

```
sub_programa encolar(cola, n, primero, ultimo, d)
      ultimo = (ultimo+1) mod n
      if  ultimo = primero  then
            cola_llena
      end(if)
      cola(ultimo) = d
fin(encolar)


sub_programa desencolar(cola, n, primero, ultimo, d)
      if primero = ultimo then
            cola_vacia
      end(if)
      primero = (primero+1) mod n
      d = cola(primero)
fin(desencolar)
```

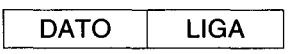
Es importante notar que la condición de cola llena y cola vacía es la misma, con la diferencia de que en el subprograma **encolar** se chequea la condición después de incrementar **ultimo**. Si la condición resulta verdadera invoca el

subprograma COLA_LLENA cuya función será dejar **ultimo** con el valor que tenía antes de incrementarla y detener el proceso, ya que no se puede encolar.

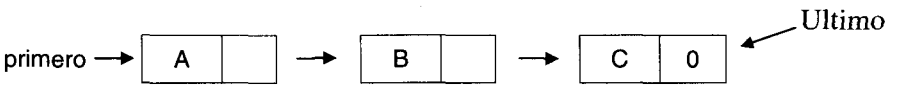
Por consiguiente habrá una posición del vector que no se utilizará, pero que facilita el manejo de las condiciones de cola vacía y cola llena.

Representación de colas como listas ligadas

Siempre que se desee representar algún objeto como lista ligada, lo primero que debe hacerse es definir la configuración del registro.

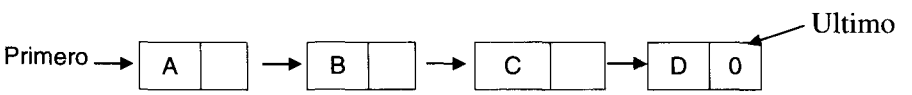


La condición de cola vacía es **primero = 0**.



Las operaciones sobre la cola son *Encolar y Desencolar.*

Encolar: Consiste en insertar un registro al final de una lista ligada.



Si la cola es vacía el dato a encolar será simultáneamente el primero y el último.

```
sub_programa encolar(primero, ultimo, d)
        new(x)
        dato(x) = d
        liga(x) = 0
        if  primero = 0  then
                primero = x
        else
                liga(último) = x
        end(if)
        último = x
fin(encolar)
```

Desencolar: Consiste en borrar el primer registro de una lista ligada (exacto a Desapilar). Habrá que controlar si la cola está o no vacía.

```
sub_programa desencolar(primero, ultimo, d)
      if primero = 0 then
            cola-vacia
      end(if)
      d = dato(primero)
      x = primero
       primero = liga(primero)
      devolver_registro(x)
fin(desencolar)
```

9.3. MANEJO DE VARIAS PILAS Y COLAS

Manejo de dos pilas en un vector

Dos pilas las podremos manejar en un solo vector de la siguiente forma: Si **n** es el número de elementos del vector, dividimos inicialmente el vector en dos partes iguales. Llamemos **m** la variable que apunta hacia el elemento de la mitad.

La primera mitad del vector será para manejar la pila 1, y la segunda mitad del vector para manejar la pila 2.

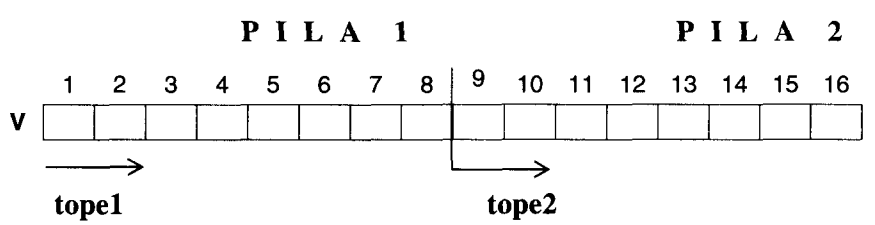
Cada pila requiere una variable tope para conocer en qué posición está el último dato de la pila. Llamemos estas variables **tope1** y **tope2** para manejar las pilas 1 y 2 respectivamente.

Consideremos el siguiente ejemplo:

Sea **V** el vector en el cual manejaremos las dos pilas.

El valor de **N** es 16.

Inicialmente el valor de **m** es 8. La mitad de **n**.



La variable **tope1** variará desde 1 hasta **m**.
La variable **tope2** variará desde **m+1** hasta **n**.
La pila 1 estará vacía cuando **tope1** sea igual a cero.
La pila 1 estará llena cuando **tope2** sea igual a **m**.
La pila 2 estará vacía cuando **tope2** sea igual a **m**.
la pila 2 estará llena cuando **tope2** sea igual a **n**.

Teniendo definido el diseño, lo que sigue es elaborar los algoritmos para manipular las pilas con él.

Consideremos primero un algoritmo para apilar un dato en alguna de las dos pilas. Habrá que especificar en cuál pila es que se desea apilar. Para ello utilizaremos un suiche (sw), el cual enviamos como parámetro del subprograma. Si el suiche es igual a 1 hay que apilar en la pila 1, si el suiche es igual a 2, hay que apilar en la pila 2.

El algoritmo es el siguiente:

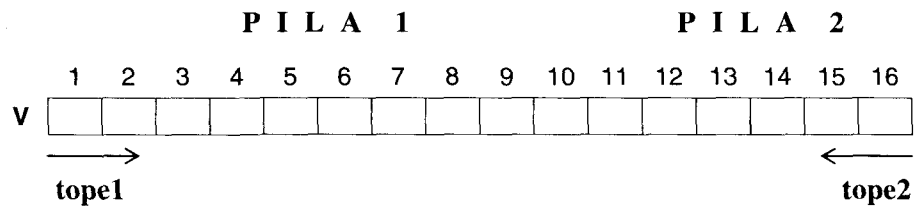
```
sub_programa apilar(v, m, n, tope1, tope2, sw, d)
      if sw = 1 then
            if tope1 = m then
                  pila_llena(sw)
            end(if)
            tope1 = tope1 + 1
            v(tope) = d
      else
            if tope2 = n then
                  pila_llena(sw)
            end(if)
            tope2 = tope2 + 1
            v(tope2) = d
      end(if)
fin(apilar)
```

El subprograma PILA_LLENA recibe como parámetro el valor del suiche.

La tarea de PILA_LLENA es: si es la pila 1 la que está llena buscará si la pila 2 está llena, en caso de no estarlo moverá los datos de la pila 2 una posición hacia la derecha, actualizará **m** y regresa al subprograma de apilar para apilar en la pila 1; si es la pila 2 la que está llena buscará si hay espacio en la pila 1, en cuyo caso moverá los datos de la pila 2 una posición hacia la izquierda, actualizará **m** y regresa a apilar en la pila 2.

Como se podrá observar la operación de apilar implica mover datos en el vector, debido a que una pila podrá crecer más rápido que la otra.

Una mejor alternativa de diseño es la siguiente:



La pila 1 se llenará de izquierda a derecha y la pila 2 de derecha a izquierda.

En este segundo diseño no hay que preocuparse por cuál pila crezca más rápidamente. Las condiciones a controlar son:

La pila 1 estará vacía cuando **tope1** sea igual a cero.
La pila 2 estará vacía cuando **tope2** sea igual a **n+1**.
Las pilas estarán llenas cuando **tope1 + 1** sea igual a **tope2**.
el algoritmo para este segundo diseño es:

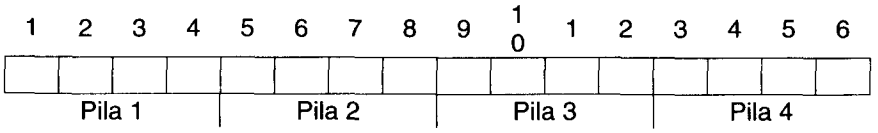
```
sub_programa apilar(v, sw, n, m, tope1, tope2, d)
      if  tope1 + 1 = tope2  then
            pila_llena
      end(if)
      if  sw = 1  then
            tope1 = tope1 + 1
            v(tope) = d
      else
            tope2 = tope2 - 1
            v(tope2) = d
      end(if)
fin(apilar)
```

Manejo de N pilas en un vector de M elementos

Ocurre muchas veces la necesidad de manejar más de dos pilas. Para la representación definida, si se desean manejar **n** pilas, una forma podría ser definir **n** vectores y **n** variables **tope**, una por cada pila que se desee manejar, sin embargo, esta forma no es la más aconsejable, ya que es impráctico e ineficiente escribir código para controlar **n** vectores y **n** variables, sin contar conque los programas

desarrollados serían particulares para el número de pilas que se deseen manejar, y que debido a que las pilas no se llenan a la misma velocidad ni uniformemente, el hecho de que una pila se llene detendría el proceso, sabiendo que en las otras pilas hay espacio para almacenar más datos.

Veremos un método para manejar **n** pilas en un vector de **m** elementos. Inicialmente se dividirá el vector en partes iguales, en tanto sea posible; serán tantas partes como pilas se deseen manejar. Consideremos el siguiente vector, que llamaremos **V**:



En nuestro ejemplo **m** vale 16 y **n** vale 4, es decir, vamos a representar 4 pilas en un vector de 16 elementos.

Necesitamos asignarle a cada pila una parte del vector, en este caso a cada pila le corresponderán 4 posiciones: la pila 1 desde la posición 1 hasta la 4, la pila 2 desde la posición 5 hasta la 8, la pila 3 desde la posición 9 hasta la 12 y la pila 4 desde la posición 13 hasta la 16.

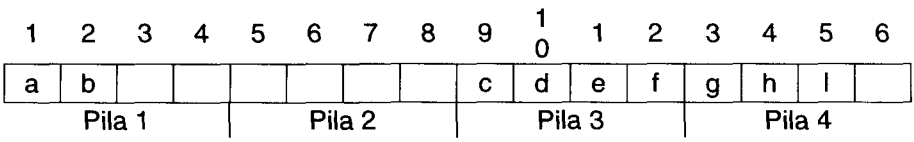
Necesitamos conocer en qué posición del vector comienza cada pila, para ello utilizaremos un vector que llamaremos **bases**, el cual en la posición **i** contendrá la posición anterior, en el vector **V**, a la que en realidad comienza la pila **i**. Más técnicamente, **bases(i)** permite conocer la posición en la cual comienza la pila **i** en el vector **V**.

Necesitamos conocer la posición del vector en la cual se halla el último elemento de cada pila, para ello utilizaremos un vector que llamaremos **topes.**

topes(i) dará la posición, en el vector **V**, en la cual se halla el último elemento de la pila **i**. Si vamos a manejar **n** pilas, entonces el vector de **topes** tendrá **n** elementos.

Inicialmente los vectores de **bases** y **topes** tendrán los mismos datos indicando que cada pila se halla vacía. Es decir, la condición de que la pila **i** se halla vacía es: **topes(i) = bases(i)**.

Si tenemos el vector **V** con la siguiente configuración:



Los vectores de **bases y topes** tendrán la siguiente información:

	1	2	3	4	5
BASES	0	4	8	12	16

	1	2	3	4
TOPES	2	4	12	15

La pila 3 se encuentra llena. Observe que **topes(3)** es igual a **bases(4)**. La condición de que la pila **i** se encuentra llena es: **topes(i) = bases(i+1).** Es por esta razón por la que el vector de **bases** tiene un elemento más que el vector de **topes**: se facilita el control de pila llena de la pila **n**.

Teniendo definida la representación y las condiciones de pila vacía y pila llena desarrollemos los algoritmos para dar las condiciones iniciales y para apilar y desapilar de cualquier pila.

Las condiciones iniciales se definen conocidos **m** y **n**, mediante las siguientes instrucciones:

```
p = m / n
for i = 1 to n do
      topes(i) = (i-1)*p
      bases(i) = (i-1)*p
end(for)
bases(n+1) = m
```

El subprograma para apilar en la pila **i** es:

```
sub_programa apilar(v, topes, bases, i, d)
      if topes(i) = bases(i+1) then
            pila_llena(i)
      end(if)
      topes(i) = topes(i) + 1
      v(topes(i)) = d
fin(apilar)
```

Donde **v** es el vector, **i** la pila en la cual se desea apilar y **d** el dato a apilar.
El subprograma para desapilar de la pila **i** es:

```
sub_programa desapilar(v,topes,bases,i,d)
      if topes(i) = bases(i) then
            pila_vacia(i)
      end(if)
      d = v(topes(i))
      topes(i) = topes(i) - 1
fin(desapilar)
```

Donde **v** es el vector, **i** la pila de la cual se va a desapilar y **d** el dato desapilado.

En el subprograma para apilar se chequea la condición de pila llena, la cual si resulta verdadera, ocasiona la ejecución del subprograma PILA_LLENA, enviando como parámetro la variable **i**, indicando que fue esta la pila que se llenó.

El subprograma PILA_LLENA buscará en las otras pilas para ver si hay espacio disponible, en cuyo caso hará los movimientos apropiados en el vector **V**, y las actualizaciones apropiadas en los vectores de **bases** y **topes** para poder apilar el dato requerido en la pila **i**.

La táctica que sigue dicho subprograma es: primero busca en las pilas a la derecha de la pila **i** (desde la pila **i+1** hasta la pila **n**), de no hallar espacio en ninguna de estas pilas buscará en las pilas a la izquierda de la pila **i** (desde la pila **i-1** hasta la pila 1), si aún no encuentra espacio, significa que todas las pilas están llenas y la operación de apilar no se puede efectuar, por tanto detendrá el proceso emitiendo un mensaje apropiado.

Veamos el algoritmo PILA_LLENA

```
sub_programa pila_llena(i)
      j = i + 1                                  // busca a la derecha de la pila i
      while j <= n and topes(j) = bases(j+1) do
            j = j + 1
      end(while)
      if j <= n then                             // encontró espacio en la pila j
            k = topes(j)
            while k > topes(i) do                // mueve los elementos
                  v(k+1) = v(k)                  //  en el vector v
                  k = k - 1
            end(while)
```

```
            for  k = i+1 to j  do                          // actualiza los
                  topes(k) = topes(k) + 1                  // vectores de topes
                  bases(k) = bases(k) + 1                  // y bases
            end(for)
            return                                  // retorna al subprograma de apilar
      end(if)
      j = i -1                                      // busca a la izquierda de la pila i
      while  j > 0  and  topes(j) = bases(j+1)  do
            j = j - 1
      end(while)
      if  j > 0  then                                 // encontró espacio en la pila j
            for  k = bases(j+1) to topes(i) - 1  do
                  v(k) = v(k+1)                       // mueve los elementos del vector v
            end(for)
            for  k = j+1 to i  do                       // actualiza los vectores
                  topes(k) = topes(k) - 1               // de topes y bases
                  bases(k) = bases(k) - 1
            end(for)
            return                                    // retorna al subprograma de apilar
      end(if)
      write('todas las pilas están llenas')
      stop
fin(pila_llena)
```

Note que el subprograma PILA_LLENA no efectúa la operación de apilar, él sólamente abre espacio y retorna al subprograma APILAR para que éste haga dicha operación.

Manejo de n pilas como listas ligadas

Con **n** pilas deberán tenerse **n** listas (una por cada pila). Para no tener que manejar **n** variables **tope**, utilizaremos un vector de apuntadores que llamaremos **topes. topes(i)** apunta hacia el primer registro de la lista que representa la pila **i**. Ver figura 9.1.

Los subprogramas para apilar y desapilar son similares a los presentado anteriormente para una pila. Basta con reemplazar **tope** por **topes(i)**.

```
sub_programa apilar(topes, i, d)
      new(x)
      dato(x) = d
      liga(x) = topes(i)
```

```
        topes(i) = x
fin(apilar)

sub_programa desapilar(topes, i, d)
        if topes(i) = 0 then
                pila_vacía(i)
        end(if)
        d = dato(topes(i))
        x = topes(i)
        topes(i) = liga(topes(i))
        devolver_registro (x)
fin(desapilar)
```

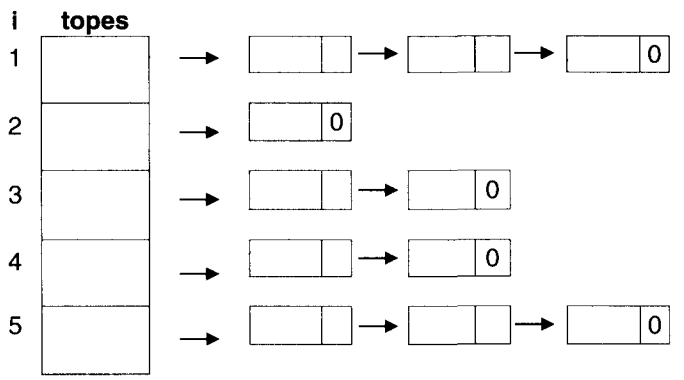
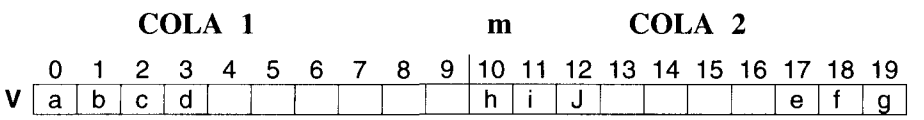


Figura 9.1

Estos subprogramas son mucho más eficientes que manejando la pila en un vector. La principal diferencia estriba en que en el subprograma para apilar se evita el proceso de pila_llena.

Manejo de dos colas en un vector de n elementos



Veamos ahora cómo trabajar dos colas, cada una circularmente, en un vector de **n** elementos. En nuestro ejemplo **n = 20**.

Inicialmente, a cada cola le corresponderá una mitad del vector. La cola 1 se representará desde la posición cero hasta la posición **m – 1**, y la cola 2 se representará desde la posición **m** hasta la posición **n – 1**.

Llamemos **p1** y **u1** las variables para identificar las posiciones en las cuales se hallan el primero y el último de la cola 1, y **p2** y **u2** las variables para identificar las posiciones en las cuales se hallan el primero y el último de la cola 2.

Las operaciones para encolar y desencolar en la cola 1 son idénticas al manejo circular de una cola en un vector:

```
p1 = (p1 + 1) % m
u1 = (u1 + 1) % m
```

Para encolar y desencolar en la cola 2, y que ésta se comporte circularmente, debemos tener en cuenta que la porción de vector que le corresponde va desde **m** hasta **n – 1**.

El número de elementos en ella es **m – n**, y para que su comportamiento sea circular debemos, temporalmente, convertir los valores de **p2** y **u2** a la forma desde 0 hasta **m – n**.

Para lograr esto, basta con restarle **m** a los valores de **p2** y **u2** antes de aplicar la operación módulo y después incrementar el resultado en **m** para que el valor obtenido quede en el rango correcto: desde **m** hasta **n – 1**.

Las instrucciones para incrementar **p2** y **u2** son:

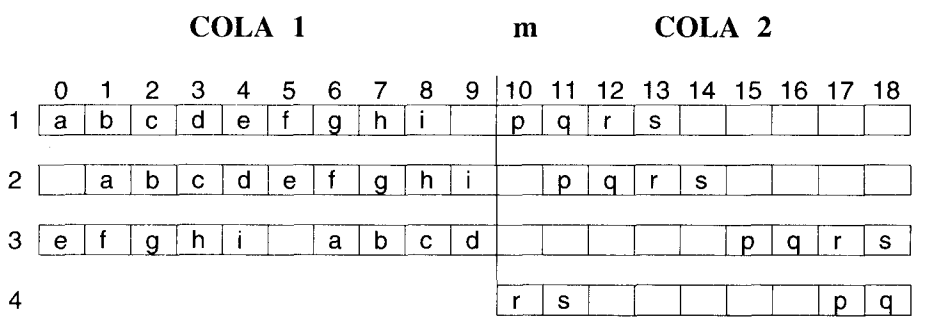
```
p2 = (p2 – m + 1) % (n – m) + m
u2 = (u2 – m + 1) % (n – m) + m
```

Teniendo definido la forma de incremento de las variables de manejo de cada cola, nuestros algoritmos para encolar y desencolar en cada una de ellas serán:

```
sub_programa encolar(V, m, n, p1, u1, p2, u2, cola, d)
    if cola = 1 then
        u1 = (u1 + 1) % m
        if u1 = p1 then
            cola_llena(cola)
        end(if)
        V(u1) = d
    else
        u2 = (u2 – m + 1) % (m – n) + m
```

```
            if  u2 = p2  then
                  cola_llena(cola)
            end(if)
            V(u2) = d
      end(if)
fin(encolar)

sub_programa desencolar(V, m, n, p1, u1, p2, u2, cola, d)
      if  cola = 1  then
            p1 = (p1 + 1) % m
            if  u1 = p1  then
                  write('cola 1 vacía')
            else
                  d = V(p1)
            end(if)
      else
            p2 = (p2 – m + 1) % (m – n) + m
            if  u2 = p2  then
                  write('cola 2 vacía')
            else
                  d = V(p2)
            end(if)
      end(if)
fin(desencolar)
```

Consideremos el algoritmo cola_llena



El subprograma cola_llena se invoca cuando se desee encolar en una de las dos colas y ésta está llena. En general, el subprograma cola_llena deberá averiguar si hay espacio disponible en la otra cola y efectuar los movimientos apropiados para abrir espacio en la cola que se llenó y poder encolar.

Comencemos considerando la situación en la cual es la cola 1 la que está llena, la cola 2 no está llena y necesitamos abrir espacio para poder encolar en la cola 1.

Para lograr este objetivo debemos hacer algún tratamiento tanto a la cola 1 como a la cola 2, dependiendo de las situaciones de cola llena y de espacio disponible respectivamente

Si la **cola 1** está llena se pueden presentar tres formas diferentes, las cuales llamaremos 1, 2 y 3. En cualquier situación tenemos **p1 = u1**.

En la situación 1, la cual identificamos porque **p1** y **u1** son iguales a **m – 1**, la acción a tomar es simplemente incrementar **p1** en 1.

En la situación 2, la cual identificamos porque **p1** y **u1** son iguales a cero, la acción a tomar es simplemente hacer **u1** igual a **m**.

En la situación 3, la cual identificamos porque **p1** y **u1** están entre **1** y **m – 2**, hay que mover los datos desde la posición **p1 + 1** hasta la posición **m – 1** una posición hacia la derecha y actualizar **p1**.

Las instrucciones correspondientes a estas situaciones son:

```
casos
    :p1 = m1:                              // situación 1
          p1 = p1 + 1
    :u1 = 0:                               // situación 2
          u1 = m
    :else:                                 // situación 3
          for i = p1+1 to m – 1 do
              v(i – 1) = v(i)
          end(for)
fin(casos)
```

Las situaciones de espacio disponible en la cola 2 las llamaremos 1, 2, 3, y 4.

En situaciones 1 y 4, las cuales identificamos porque **p2** es mayor que **u2**, hay que mover los datos desde la posición **m** hasta la posición **u2** una posición hacia la derecha y actualizar **u2**.

La situación 2, la cual identificamos porque **p2** es igual a **m** basta con asignar a **p2** el valor de **n – 1**. Pero antes de hacer esto debemos considerar el hecho de que de pronto la cola 2 está vacía y entonces habrá que asignar **n – 1** tanto a **p2** como a **u2**.

En la situación 3 no hay que efectuar operación alguna.

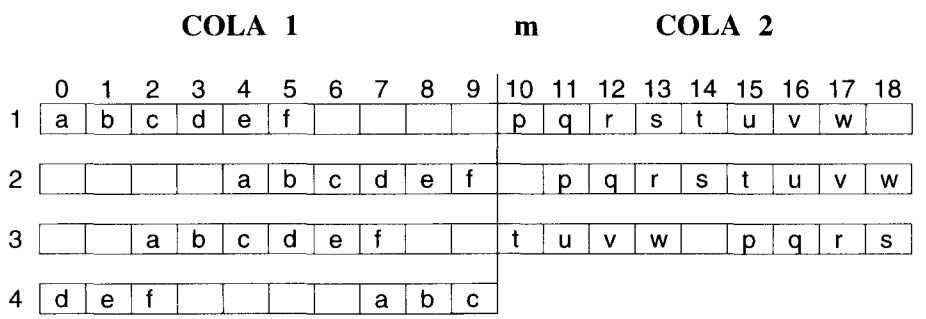
Las instrucciones correspondientes son:

```
casos
      :p2 > u2:
            for i = u2 downto m do
                V(i + 1) = V(i)
            end(for)
            u2 = u2 + 1
      :p2 = m:
            if p2 = u2 then
                u2 = n - 1
            end(if)
            p2 = n - 1
fin(casos)
```

Cualquiera que hubieran sido los tratamientos que se hayan hecho sobre ambas colas el valor de **m** hay que incrementarlo en 1.

Analicemos ahora el caso en que hubiera sido la cola 2 la que está llena y la cola 1 tiene espacio.

Consideremos las diferentes situaciones para este caso.



Las situaciones de la cola 2 llena son 3, las cuales identificaremos como 1, 2 y 3. En cualquier situación **p2 = u2**.

La situación 1, la cual identificamos porque **p2 = n – 1**, basta con hacer **p2 = m – 1**

La situación 2, la cual identificamos porque **u2 = m**, basta con hacer **u2 = m –1**.

La situación 3 es cuando no se cumple ninguna de las dos condiciones anteriores y será necesario mover los datos en el vector desde la posición **m** hasta la posición **u2** una posición hacia la izquierda.

Las instrucciones correspondientes son:

```
casos
      :p2 = n – 1:
            p2 = m – 1
      :u2 = m:
            u2 = m – 1
      :else:
            for i = m to u2 do
                V(i – 1) = V(i)
            end(for)
fin(casos)
```

Consideremos ahora el tratamiento a la cola 1. Hay 4 situaciones de cola 1 con espacio:

Situaciones 1 y 4, las cuales identificamos porque **p1 > u1** se solucionan así: si **p1** es igual a **m – 1** basta con restar 1 a **p1**, de lo contrario hay que mover los datos en el vector, desde la posición **p1** hasta la posición **m – 1**, una posición hacia la izquierda y restarle 1 a **p1**.

Situaciones 2 y 3, las cuales identificamos porque **p1 < u1** se solucionan así: si **u1** es menor que **m – 1** no hay que tomar ninguna acción, de lo contrario hay que mover todos los datos de la cola uno una posición hacia la izquierda y restar uno a **p1** y a **u1**.

Cualquiera que hubieran sido los tratamientos a las dos colas hay que restar 1 a **m**.

Las instrucciones correspondientes son;

```
casos
    :p1 > u1:
          if p1 < m – 1 then
              for i = p1 to m – 2 do
                  V(i) = V(i+1)
              end(for)
          end(if)
```

```
            p1 = p1 – 1
    :p1 < u1:
            if  u1 = m – 1   then
                    for  i = p1  to  m – 2   do
                            V(i) = V(i+1)
                    end(for)
                    p1 = p1 – 1
                    u1 = u1 – 1
            end(if)
fin(casos)
```

Agrupando todas las situaciones planteadas nuestro algoritmo de cola_llena queda así:

```
Subprograma cola_llena(cola)
    if  cola = 1  then
      if  ((u2 – m + 1) % (m – n) + m) <> p2  then   // hay espacio en cola 2
                                                     // tratamiento cola 1
          casos
              :p1 = m1:
                    p1 = p1 + 1
              :u1 = 0:
                    u1 = m
              :else:
                    for  i = p1+1  to  m – 1   do
                        V(i – 1) = V(i)
                    end(for)
          fin(casos)
                                                       // tratamiento cola 2
          casos
              :p2 > u2:
                     for  i = u2  downto  m   do
                         V(i + 1) = V(i)
                    end(for)
                    u2 = u2 + 1
              :p2 = m:
                    if  p2 = u2  then
                        u2 = n – 1
                    end(if)
                    p2 = n – 1
          fin(casos)
          m = m + 1
          return
```

```
      end(if)
    else
      if (u2 + 1) % m <> p2  then              // hay espacio en cola 1
        casos                                  // tratamiento cola 2
          :p2 = n – 1:
                p2 = m – 1
          :u2 = m:
                u2 = m – 1
          :else:
                for i = m to u2 do
                   V(i – 1) = V(i)
                end(for)
        fin(casos)
        casos
          :p1 > u1:
                if p1 < m – 1 then
                  for i = p1 to m – 2 do
                     V(i) = V(i+1)
                  end(for)
                end(if)
                p1 = p1 – 1
          :p1 < u1:
                if u1 = m – 1 then
                   for i = p1 to m – 2 do
                      V(i) = V(i+1)
                   end(for)
                   p1 = p1 – 1
                   u1 = u1 – 1
                end(if)
        fin(casos)
        m = m – 1
        return
    end(if)
    write('ambas colas están llenas')
    stop
fin(cola_llena)
```

Manejo de N colas en un vector

De la misma forma que puede suceder que haya que manejar **n** pilas, también puede suceder que haya que manejar **n** colas. De una forma similar podemos manejar **n** colas en un vector de **m** elementos. En el manejo de colas necesitaríamos tres vectores auxiliares: uno que indique en cuál posición del vector empieza cada

cola, otro que indique en cuál posición del vector se halla el primer elemento de cada cola y otro que indique en cuál posición del vector se halla el último elemento de cada cola.

Manejo de N colas como listas ligadas

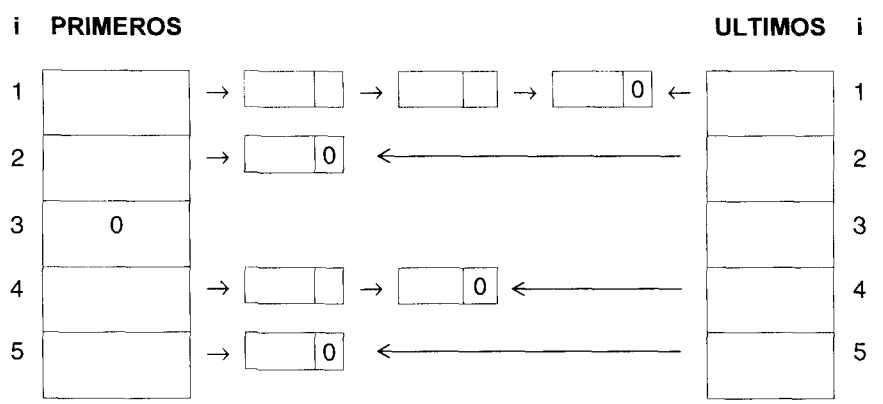
Equivale a manejar **n** listas, una lista por cada cola.

Para manejar **n** colas como listas ligadas simplemente manejamos dos vectores, los cuales llamaremos primeros y ultimos.

Primeros(i): Apunta hacia el primer registro de la lista ligada que representa la cola **i**.

Ultimos(i): Apunta hacia el último registro de la lista ligada que representa la cola **i**.

Los subprogramas para ENCOLAR Y DESENCOLAR son los mismos que manejando una sola cola, con la inclusión del parámetro **i**, el cual indica en cuál cola voy a encolar (o desencolar), y cambiando PRIMERO por PRIMEROS(i) y ULTIMOS por ULTIMOS(i).



Representación gráfica cada una como lista ligada del manejo de N colas.

EJERCICIOS PROPUESTOS

1. Elabore un algoritmo que trabaje una cola circular utilizando todos los elementos del vector.

2. Elabore un algoritmo que maneje una pila y una cola en un vector. La cola se debe manejar circularmente.

3. Elabore un algoritmo que maneje dos colas en un vector. Escriba algoritmos para encolar, desencolar, y cola_llena.

4. Diseñe la forma de manejar N colas en un vector de M elementos. Cada cola se debe manejar circularmente. Escriba algoritmos para encolar o desencolar de alguna cola. Escriba también un algoritmo de cola llena.

5. Diseñe la forma de representar una cola como lista ligada, en la cual sólo utilice un apuntador de entrada a la lista. Su diseño debe ser tal que los algoritmos de encolar y desencolar sean O(1).

10

RECURSIÓN

Es la técnica mediante la cual se define una función o un proceso en términos de sí mismo. Es una poderosa herramienta que no ha tenido la difusión ni la acogida que debiera. Hay básicamente dos razones para ello: una, es que hay muchas personas que aprendieron el arte de programar computadores con lenguajes que no admiten esta técnica, por lo tanto no han podido disfrutar de sus beneficios; dos, que los algoritmos recursivos tienen una sobrecarga de tiempo en la ejecución. Sin embargo, esto no debe ser obstáculo para utilizar la recursión debido a que hay procedimientos para convertir algoritmos recursivos a versiones no recursivas e incrementar la eficiencia de éstos. Además la recursión ha tenido el mito de la complejidad, idea poco aceptable, debido a que quizá es más sencillo y más legible, en muchos casos, plantear algoritmos recursivos que su versión iterativa alterna.

Digamos también, que independientemente de los computadores, esta es una técnica que los matemáticos utilizan muy frecuentemente en la definición de sus funciones. Un ejemplo clásico es la función factorial. Según los matemáticos:

$$n! = \begin{cases} 1 \text{ si } n = 0, \\ n*(n-1)! \text{ si } n > 0 \end{cases}$$

Esta es una definición recursiva ya que define el factorial de **n** en términos del factorial de **n-1**.

Los objetivos que perseguimos en este capítulo son:

- Aprender a pensar recursivamente.
- Convertir ciclos en algoritmos recursivos.
- Hacer seguimiento a algoritmos recursivos.
- Analizar y plantear algoritmos recursivos.
- Convertir algoritmos recursivos a versiones no recursivas.

Para lograr el primer objetivo es interesante ver cómo las operaciones aritméticas se pueden plantear en forma recursiva.

Empecemos considerando el caso de la exponenciación. Se deseamos efectuar 4 elevado a la potencia 5, recordemos que la potenciación es una sucesión de multiplicaciones, por tanto podremos escribir:

$4^5 = 4*4*4*4*4$

lo cual también podremos escribir así: $4^5 = 4*(4*4*4*4)$

que es lo mismo que $4^5 = 4*4^4$

y ahí tenemos cuatro a la cinco expresado en términos de cuatro a la cuatro.

Esto es recursión, definir algo en términos de sí mismo.

De una manera análoga podemos seguir definiendo cuatro a la cuatro:

Cuatro a la cuatro es cuatro por cuatro a la tres.
Cuatro a la tres es cuatro por cuatro a la dos.
Cuatro a la dos es cuatro por cuatro a la uno.
Cuatro a la uno es cuatro por cuatro a la cero.

Y cuatro a la cero es uno por definición: todo número elevado a la potencia cero es uno.

De acuerdo al anterior análisis podemos sintetizar la operación de exponenciación de la siguiente manera:

$$n^r = \begin{cases} 1 & \text{si } r = 0, \\ n*n^{r-1} & \text{si } r > 0 \end{cases}$$

Para continuar nuestro estudio sobre recursión, y con miras a que el estudiante no adquiera temor con esta técnica, diremos que cualquier ciclo: **for, while, repeat** se puede reescribir como un algoritmo recursivo.

Plantearemos el ejemplo con el ciclo **for**. Para el ciclo **while** la técnica es exactamente la misma; para el ciclo **repeat** se debe convertir primero a un ciclo **while**, y esto siempre es posible.

10.1 PASOS PARA CONVERTIR UN CICLO FOR EN UN SUBPROGRAMA RECURSIVO

1. Reemplazar el ciclo **for** por sus instrucciones elementales.
1.1 Asignar valor inicial a la variable controladora del ciclo.
1.2 Escribir una instrucción **if** que compare el valor de la variable controladora del ciclo con el valor final.
1.3 Rotular el **if** escrito en 1.2
1.4 Escribir las instrucciones propias del ciclo.
1.5 Incrementar la variable controladora del ciclo.
1.6 Transferir el control **(goto)** al label definido en 1.3.
1.7 Cerrar el **if** definido en 1.2 **(end)**.
2. El subprograma recursivo lo conformarán todas las instrucciones del ciclo y su parámetro es la variable controladora del ciclo.
3. En el programa llamante se suprimen todas las instrucciones del ciclo y se reemplaza la instrucción de asignación del valor inicial a la variable controladora del ciclo, por una llamada al subprograma recursivo. El valor del parámetro en esta llamada es el valor inicial de la variable controladora del ciclo.
4. En el subprograma recursivo se reemplazan las instrucciones de incremento de la variable controladora del ciclo y de transferencia de control **(goto)** por una llamada recursiva, con parámetro la variable controladora del ciclo modificada.

Ejemplo: sea un programa principal:

```
read(n,x)
for i = 1 to n do
x= x+1
end(for)
write(x)
```

n y **x** son variables globales.

Paso 1: Reemplazar el ciclo FOR por sus instrucciones elementales:

```
      read(n, x)
      i = 1
:lb:   if i <= n then
          x = x + 1
          i = i + 1
          goto lb
       end(if)
       write(x)
```

Paso 2: El subprograma recursivo lo conforman:

```
:lb:  if  i <= n then
          x = x + 1
          i = i + 1
           goto lb
      end(if)
```

Paso 3: El programa principal queda así:

```
read(n, x)
sr(1)
write(x)
```

Paso 4: El subprograma recursivo queda:

```
sub_programa sr(i)
    if   i <= n then
         x = x + 1
         sr(i+1)
      end(if)
fin(sr)
```

Consideremos otro ejemplo con dos ciclos FOR anidados

```
read(k, l)
n = k
m = k
for  j = 1 to l -1  do
     for  i = 1 to k-1  do
          m = m + n
     end(for)
     n = m
end(for)
write(k,l,n)
```

k, l, m, n son variables globales.

Paso 1: reemplazar los ciclos FOR por sus instrucciones elementales:

```
      read(k, l)
      n = k
      m = k
      j = 1
:l1:   if  j <= l – 1  then
           i = 1
:l2:       if  i <= k-1 then
```

```
            m = m + n
            i = i + 1
          goto l2
        end(if)
        n = m
        j = j + 1
        goto l1
      end(if)
      write(k, l, n)
```

Paso 2: el subprograma recursivo externo es:

```
:l1:  if  j <= l – 1  then
        i = 1
:l2:    if  i <= k – 1  then
          m = m + n
          i = i + 1
          goto l2
        end(if)
        n = m
        j = j + 1
        goto l1
    end(if)
```

Paso 3: el programa principal queda así:

```
    read(k, l)
    n = k
    m = k
    s1(1)
    write(k, l, n)
```

Paso 4: el subprograma recursivo S1 es:

```
  sub_programa s1(j)
        if  j  <=  l –1   then
              i = 1
  :l2:        if  i  <=  k –1   then
                    m = m + n
                    i = i + 1
              goto  l2
              end(if)
              n = m
              s1(j+1)      /* llamada recursiva */
        end(if)
  fin(s1)
```

El cual queda con las instrucciones elementales del ciclo interno, las cuales, al aplicarles los mismos tres primeros pasos, obtenemos los siguientes procedimientos:

```
sub_programa s1(j)
      if j <= l-1  then
            s2(1)              /* llamada al procedimiento del ciclo interno */
            n = m
            s1(j+1)            /* llamada recursiva */
      end(if)
fin(s1)


sub_programa s2(i)
      if i <= k-1 then
            m = m + n
            s2(i+1)            /* llamada recursiva */
      end(if)
fin(s2)
```

Ya hemos visto cómo convertir un ciclo en un subprograma recursivo. Ocupémonos ahora de ver cómo funcionan dichos algoritmos recursivos. Para ello vamos a ver cómo se hace el seguimiento a un algoritmo recursivo.

Antes de entrar en materia es necesario tener claro los siguientes conceptos:

Variable global: es una variable definida externamente al subprograma recursivo y que es trabajada y modificada dentro del subprograma. Toda modificación que sufra dentro del subprograma afecta su valor externo.

Variable local: es una variable definida dentro del subprograma. Nace y muere dentro del subprograma. Es independiente del programa llamante o principal.

Parámetros del subprograma: son los datos que recibe el subprograma. Hay básicamente dos formas de recibir dichos datos: por **valor**, o por **variable o referencia.**

Parámetros por valor: son datos o variables que recibe el subprograma. Cualquier modificación que sufran estas variables dentro del subprograma es vigente sólo dentro del subprograma, es decir, el contenido externo de la variable no sufre ninguna modificación.

Parámetros por variable o referencia: son datos que recibe el subprograma en nombres de variables. Toda modificación que se haga dentro del subprograma al contenido de estas variables afecta la variable en el programa llamante. Se comportan como si fueran variables globales.

Dirección de retorno: se refiere a la instrucción donde debe continuar la ejecución de un programa o subprograma cuando termina la ejecución de otro subprograma que fue invocado desde el primero.

En esto de las direcciones de retorno hay que distinguir entre los dos tipos de subprogramas que existen: Funciones y Procedimientos.

Recordemos que las funciones retornan un solo valor, el cual regresa en el nombre de la función, o sea que éste se comporta como una variable. Los procedimientos ejecutan alguna tarea específica y el nombre de ellos es simplemente un nombre para poderlos referenciar.

Debido a esto, la dirección de retorno para las funciones es la misma instrucción que contiene la llamada a la función, mientras que para los procedimientos la dirección de retorno es la instrucción siguiente a la instrucción que hizo la llamada al procedimiento.

La recursión se presenta cuando dentro de las instrucciones de un subprograma existe una o más llamadas a la ejecución del mismo subprograma.

Debido entonces, a que antes de terminarse la ejecución del subprograma se reinicia de nuevo su ejecución, y así sucesivamente, se podría pensar que la ejecución del subprograma se repite indefinidamente; es por esto por lo que cuando se presenta recursión, la llamada recursiva debe estar sujeta a alguna condición. Cuando esta condición no se cumple no habrá llamada recursiva y la ejecución del subprograma de alguna forma terminará.

Considerando el caso general, un subprograma (FUNCTION o PROCEDURE) tiene uno o más parámetros de llamada. Dependiendo del valor de uno o varios de esos parámetros se hace llamada recursiva o no. Esta llamada recursiva enviará nuevos valores de los parámetros con los cuales se determinará si hay o no nuevas llamadas recursivas. Cuando ocurra que no se hizo llamada recursiva el subprograma terminará la ejecución de esa llamada y, como en la ejecución de cualquier subprograma, el control retornará al programa llamante (en este caso el mismo subprograma), al sitio apropiado, a continuar con una ejecución interrumpida.

Cuando se inicia la ejecución de un subprograma recursivo, es decir, cuando es invocado por primer vez, se asignan unos valores iniciales a los parámetros del subprograma. A medida que avanza la ejecución se van asignando y/o alterando los valores correspondientes a los parámetros y a algunas variables propias del subprograma (variables locales), y dependiendo de alguna condición se hará o no llamada recursiva. Cuando se hace la llamada recursiva, los parámetros y variables locales tienen algún valor almacenado, los cuales hay que conservar para que cuando se regrese a continuar con la primera llamada, o sea, cuando se termine con la llamada recursiva, se continúe con los valores correctos.

Debido a que la llamada recursiva se puede ejecutar un sinnúmero de veces, por cada una de ellas hay que conservar los valores de variables locales y parámetros correspondientes a esa llamada, para que cuando regrese a continuarla, lo haga con los valores correctos.

La forma de recuperar estos datos es en forma inversa a la que fueron guardados, por lo tanto, la estructura apropiada para guardarlos es una pila. Por consiguiente, para hacer seguimiento a un algoritmo recursivo utilizaremos una pila.

Resumiendo todo lo anterior, los pasos para hacer seguimiento a un algoritmo recursivo son los siguientes:

1. Definir una pila vacía.
2. Definir dirección(es) de retorno.
3. Cada vez que se encuentre una llamada recursiva guardar en la pila:
   - parámetros del subprograma,
   - variables locales,
   - dirección de retorno y
   - nombre de la función, si se trata de una función.
4. Asignar los nuevos valores de llamada a los parámetros.
5. Comenzar de nuevo la ejecución con los nuevos datos.
6. Cada vez que se termine una ejecución sacar de la pila los datos correspondientes a una llamada recursiva y regresar a la dirección de retorno desapilada a continuar la ejecución con los datos que se sacaron de la pila.
7. Cuando no haya más datos en la pila es señal de que se terminó el proceso y se regresa al programa llamante a continuar su ejecución.

Ilustraremos el proceso con el ejemplo clásico de la función factorial. Como habíamos visto la función factorial se define así:

$$n! = \begin{cases} 1 \text{ si } n = 0, \\ n*(n-1)! \text{ si } n > 0 \end{cases}$$

A partir de esta definición elaborar el algoritmo es cuestión de escribir lo mismo en notación algorítmica:

El subprograma (FUNCTION) para determinar el factorial de un número N es:

```
1     function fact(n)
2           if n = 0 then
3               fact = 1
4           else
5  l1 →         fact = n*fact(n-1)
6           end(if)
7     fin(fact)
```

El primer paso es definir una pila vacía, la cual contendrá los datos correspondientes a N (parámetro de la función), la dirección de retorno que llamaremos L1 y que corresponde a la instrucción 5 y el nombre de la función, ya que cuando se trate de una función, su valor se guardará también en la pila y se calculará a medida que se vayan resolviendo las llamadas recursivas.

La dirección de retorno es la instrucción 5 ya que se trata de una función y es en esta instrucción donde se halla la llamada recursiva.

Supongamos entonces que existe un programa principal en el cual tenemos el siguiente conjunto de instrucciones:

```
        _____
        _____
L0 →  M = FACT(4)
        _____
        _____
```

Llamamos L0 la dirección de retorno al programa llamante. La primera llamada se efectúa con N = 4. El seguimiento comienza así:

L0	4	?
DR	N	FACT

Lo cual significa que se va a ejecutar la FUNCTION FACT con N valiendo 4 y cuando se termine se regresará a la instrucción que llamamos L0.

Comenzando la ejecución de la función, con N=4, se pregunta si N = 0, lo cual es falso, luego ejecuta la instrucción 5 que es la que contiene la llamada recursiva. La pila quedará así:

L1	3	?
L0	4	?
DR	N	FACT

indicando que se interrumpe la ejecución de la función con N=4, y que se va a ejecutar la función FACT con N=3 y que cuando se termine esta ejecución se regresará a la instrucción llamada L1 a continuar la ejecución cuando N=4.

Comenzando, de nuevo, la ejecución de la función con N=3, N es diferente de cero y ejecuta de nuevo la instrucción 5, que es la que contiene la llamada recursiva, por tanto se interrumpe la ejecución de la función con N=3 y el estado de la pila queda así:

L1	2	?
L1	3	?
L0	4	?
DR	N	FACT

indicando que se va a ejecutar la FUNCTION FACT con N=2 y que cuando termine dicha ejecución regresará a la instrucción llamada L1 a continuar la ejecución de la función FACT con N=3.

Razonando igualmente para la ejecución con N=2 y N=1 la pila queda así:

L1	0	?
L1	1	?
L1	2	?
L1	3	?
L0	4	?
DR	N	FACT

L1	0	1
L1	1	?
L1	2	?
L1	3	?
L0	4	?
DR	N	FACT

Ahora debe retornar a la instrucción llamada L1 con FACT valiendo 1 y desapilar todos los datos correspondientes a la llamada con N=0. El estado de la pila cuando retorna a esta instrucción es el siguiente:

L1	1	1
L1	2	?
L1	3	?
L0	4	?
DR	N	FACT

Multiplica el contenido de N que es 1 por el contenido de FACT que es 1 y el resultado lo deja en FACT, o sea, nuevamente 1, y continúa con la instrucción 6, la cual es el fin del IF y luego el fin de la función. Ha terminado entonces la ejecución de la función con N=1 y FACT quedó valiendo 1. Debe regresar a la instrucción llamada L1 con FACT valiendo 1 para continuar la ejecución de la función cuando N=2. El estado de la pila es ahora el siguiente antes de continuar la ejecución con N=2:

L1	2	1
L1	3	?
L0	4	?
DR	N	FACT

Multiplica el contenido de FACT, que es 1, por el contenido de N, que es 2, y el resultado lo deja en FACT quedando la pila así:

L1	2	2
L1	3	?
L0	4	?
DR	N	FACT

La instrucción siguiente es el fin del IF y la siguiente el fin de la función. Ha terminado la ejecución de la función con N=2 y debe regresar a la instrucción llamada L1 (instrucción 5) a continuar la ejecución de la función con N=3 y FACT regresó valiendo 2. La pila está así:

L1	3	2
L0	4	?
DR	N	FACT

Ejecuta la instrucción 5, el resultado lo deja en FACT y la pila queda así:

L1	3	6
L0	4	?
DR	N	FACT

Retorna nuevamente a la instrucción 5 a continuar la ejecución con N=4 y FACT=4. La pila queda así:

L0	4	6
DR	N	FACT

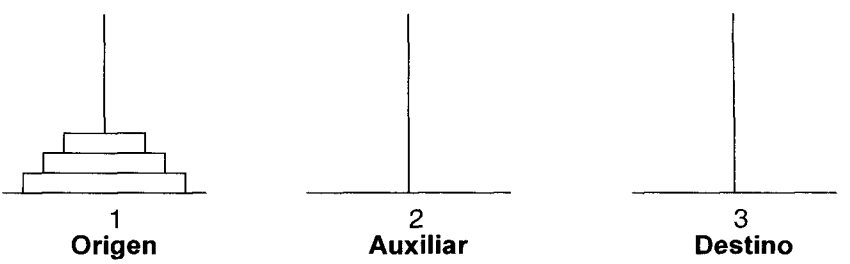
Multiplica N por FACT y el resultado lo deja en FACT. La pila queda así:

L0	4	24
DR	N	FACT

Continúa con el fin del IF y el fin de la función y retorna a la instrucción L0 del programa llamante, con FACT valiendo 24, para asignárselo a M y continuar con dicha ejecución.

10.2 TORRES DE HANOI

Consideremos otro problema clásico en materia de recursión: el algoritmo de las Torres de Hanoi. Analizaremos el problema y plantearemos su solución recursiva y escribiremos el correspondiente algoritmo. El problema en realidad es un juego que consiste en lo siguiente: se tienen 3 columnas numeradas 1, 2, y 3 y en alguna de ellas hay N aros de diferente tamaño, colocados de tal forma que el aro más grande se halla en el fondo y el aro más pequeño en la parte superior. La columna en la que se hallan los aros se denomina columna **origen**. El objetivo es trasladar los aros desde la columna origen hacia otra de las columnas, denominada **destino**, utilizando la tercera columna como **auxiliar**, cumpliendo siempre dos condiciones: sólo se puede mover un aro a la vez y no podrá nunca haber un aro de algún tamaño encima de un aro de menor tamaño. Esquemáticamente la posición inicial podría ser la siguiente:



En la columna 1 tenemos 3 aros, los cuales debemos pasar hacia la columna 3, utilizando la columna 2 como auxiliar.

Nuestro programa de computador debe imprimir los movimientos apropiados para efectuar el traslado, cumpliendo las condiciones dadas.

Una secuencia de movimientos para nuestro ejemplo sería:

1. $1 \rightarrow 3$
2. $1 \rightarrow 2$
3. $3 \rightarrow 2$
4. $1 \rightarrow 3$
5. $2 \rightarrow 1$
6. $2 \rightarrow 3$
7. $1 \rightarrow 3$

La cual, si analizamos detenidamente, la podemos descomponer de esta forma:

Tenemos una torre de 3 aros (N=3),

Los primeros 3 movimientos trasladan una torre de 2 aros (N –1 aros) desde la columna **origen** hacia la columna **auxiliar**,

El movimiento 4 traslada el aro de la base de la columna **origen** hacia la columna **destino**,

Los movimientos 5 a 7 trasladan torre de 2 aros (N-1 aros) desde la columna **auxiliar** hacia la columna **destino**.

Diciéndolo de otra forma, para poder llevar el aro que está en la base de una torre de N aros, desde una columna origen hacia una columna destino tengo que trasladar primero N –1 aros desde la columna origen hacia la columna auxiliar, luego muevo el aro (imprimo el movimiento) desde origen hacia destino y por último traslado la torre de N-1 aros que tengo en la columna auxiliar hacia la columna destino. Escribiéndolo en una forma más esquemática tendremos:

Mover torre de N aros desde Origen hacia Destino usando Auxiliar es:

1. Mover torre de N – 1 aros desde Origen hacia Auxiliar.
2. Mover aro desde Origen hacia Destino.
3. Mover torre de N – 1 aros desde Auxiliar hacia Destino.

Lo cual es un planteamiento recursivo del problema, ya que estoy definiendo el proceso de mover torre en términos de mover torre y eso es recursión: definir algo en términos de sí mismo. Dicho proceso se repite hasta que no haya aros para mover, es decir, hasta que N sea igual a cero.

El algoritmo para mover torre de N aros es entonces:

```
1.      sub_programa mover_torre(n,o,a,d)
2.        if n > 0 then
3.             mover_torre(n-1,o,d,a)
4.  l1 →    write(o,'=>',d)
5.             mover_torre(n-1,a,o,d)
6.  l2 →end(if)
7.      fin(mover_torre)
```

Los parámetros del procedimiento son:

n = número de aros
o = columna origen

a = columna auxiliar
d = columna destino

Veamos ahora cómo hacer el seguimiento a este algoritmo, con una llamada para mover 2 aros desde la columna 1 hacia la columna 3 utilizando como auxiliar la columna 2. Sea este el programa llamante:

```
        ——
        ——
        mover_torre(2,1,2,3)
L0 →    ——
        ——
```

Nuestro primer paso será definir las direcciones de retorno, las cuales, como es un procedimiento serán las instrucciones siguientes a las instrucciones donde haya llamada al procedimiento MOVER_TORRE.

En el programa llamante denominaremos esta instrucción L0 y en el procedimiento MOVER_TORRE las llamamos L1 y L2 respectivamente, que corresponden a las instrucciones 4 y 6 de nuestro algoritmo MOVER_TORRE.

Utilizaremos una pila en la cual guardaremos la información correspondiente a los parámetros (N,O,A,D) y a las direcciones de retorno.

Variables locales no hay. Al ejecutarse la primera llamada nuestra pila quedará así:

L0	2	1	2	3
DR	N	O	A	D

indicando que se va a ejecutar el procedimiento MOVER_TORRE con N=2, O=1, A=2 y D=3 y que cuando termine, regresará a la instrucción llamada L0 a continuar ejecutando el programa llamante.

Al comenzar la ejecución del procedimiento MOVER_TORRE la primera instrucción pregunta si N es mayor que cero, como el resultado es verdadero ejecutará la instrucción 3, la cual es una llamada recursiva, por tanto guardará en la pila los datos correspondientes a la nueva llamada y la pila quedará así:

L1	1	1	3	2
L0	2	1	2	3
DR	N	O	A	D

indicando que va a ejecutar el procedimiento MOVER_TORRE con N=1, O=1, A=3 y D=2 y que cuando termine regresará a la instrucción rotulada L1 a continuar ejecutando MOVER_TORRE con los datos de N, O, A y D que están inmediatamente debajo en la pila.

Al comenzar a ejecutar de nuevo MOVER_TORRE, la condición N mayor que cero, es nuevamente verdadera y ejecutará de nuevo MOVER_TORRE con N=0, O=1, A=2 y D=3 y la dirección de retorno será L1. El estado de la pila será entonces el siguiente:

L1	0	1	2	3
L1	1	1	3	2
L0	2	1	2	3
DR	N	O	A	D

indicando que se va ejecutar MOVER_TORRE con N=0, O=1, A=2 y D=3 y que cuando termine regresará a la instrucción L1 a continuar ejecutando una llamada interrumpida.

Como N es igual a cero la condición de la instrucción 2 es falsa y la ejecución continúa en la instrucción siguiente al END(if), la cual es el fin del procedimiento, lo que significa que terminó de ejecutar dicha llamada y por consiguiente desapilará los datos correspondientes a esta llamada y regresará a la instrucción rotulada L1 a continuar la ejecución del procedimiento MOVER_TORRE con los datos que están inmediatamente debajo. El estado de la pila es entonces el siguiente:

L1	1	1	3	2
L0	2	1	2	3
DR	N	O	A	D

La instrucción L1 dice que imprima **O** y **D**, por tanto imprime **1 --> 2** y continúa con la siguiente instrucción que es un nuevo llamado a MOVER_TORRE, con N=0, O=3, A=1 y D=2 y dirección de retorno L2. La pila queda así:

L2	0	3	1	2
L1	1	1	3	2
L0	2	1	2	3
DR	N	O	A	D

indicando que ejecutará MOVER-TORRE con los datos que hay en el tope y que cuando termine regresará a la instrucción rotulada L2 a continuar la ejecución con los datos que están inmediatamente debajo.

La ejecución de MOVER_TORRE con estos datos ocasiona que la condición de la instrucción 2 sea falsa y la ejecución continúa en la instrucción siguiente al END(if), la cual es el fin del procedimiento, por tanto desapilará los datos correspondientes a esta llamada y regresará a la instrucción rotulada L2 a continuar la llamada anterior que se había interrumpido y cuyos datos están en el nivel inmediatamente debajo en la pila. El estado de la pila es el siguiente:

L1	1	1	3	2
L0	2	1	2	3
DR	N	O	A	D

Al regresar a la instrucción rotulada L2, ésta es el END(if), y la instrucción siguiente es el fin del procedimiento, lo que significa que terminó de ejecutar esta llamada, por tanto desapila los datos correspondientes a esta llamada y regresa a la instrucción rotulada L1 a continuar la ejecución de MOVER_TORRE con los datos que están inmediatamente debajo. El estado de la pila es el siguiente:

L0	2	1	2	3
DR	N	O	A	D

La instrucción L1 es WRITE(O,'→',D), luego escribe **1 → 3**, que es el siguiente movimiento y continúa con la instrucción siguiente al WRITE, la cual es una llamada recursiva al procedimiento MOVER_TORRE con N=1, O=2, A=1 y D=3 y dirección de retorno L2.

El estado de la pila es el siguiente:

L2	1	2	1	3
L0	2	1	2	3
DR	N	O	A	D

Al comenzar a ejecutar de nuevo el procedimiento MOVER_TORRE con estos datos, la condición de la instrucción 2 es verdadera y ejecuta la instrucción 3, la cual es una llamada a MOVER_TORRE con N=0, O=2, A=3 y D=1 y dirección de retorno L1. La pila queda así:

L1	0	2	3	1
L2	1	2	1	3
L0	2	1	2	3
DR	N	O	A	D

La ejecución de MOVER_TORRE con estos datos hacen que la instrucción 2 sea falsa y ocasionan la terminación de esta llamada, por tanto desapila los datos del tope y retorna a la instrucción L1 a continuar la llamada interrumpida cuyos datos se hallan inmediatamente debajo. El estado de la pila es el siguiente:

L2	1	2	1	3
L0	2	1	2	3
DR	N	O	A	D

La instrucción L1 dice que imprima el contenido de O y D, o sea, que imprime **'2 → 3'** y continúa con la instrucción siguiente que es otra llamada a MOVER_TORRE con N=0, O=1, A=2 y D=3 y dirección de retorno L2. El estado de la pila será entonces:

L2	0	1	2	3
L2	1	2	1	3
L0	2	1	2	3
DR	N	O	A	D

La ejecución con estos últimos datos no ocasionan una nueva llamada ya que N=0 da por terminada la ejecución del procedimiento, luego desapila estos datos y regresa a la instrucción llamada L2 a continuar ejecutando la llamada anterior. El estado de la pila es:

L2	1	2	1	3
L0	2	1	2	3
DR	N	O	A	D

La instrucción L2 es el END(if) y luego sigue el fin del procedimiento, o sea que termina esta llamada y por tanto desapila los datos correspondientes y regresa a la instrucción L2 a continuar con la llamada cuyos datos están inmediatamente debajo en la pila. El estado de la pila es:

L0	2	1	2	3
DR	N	O	A	D

La instrucción L2, según hemos visto antes, da por terminado el proceso, entonces desapila y retorna a la instrucción rotulada L0 a continuar ejecutando el programa llamante.

Es muy importante que el lector haya entendido perfectamente el anterior seguimiento para que pueda avanzar en el tema.

10.3 CONVERSION DE ALGORITMO RECURSIVO EN ALGORITMO NO RECURSIVO

Realmente, si se tiene clara la forma de hacer seguimiento a un algoritmo recursivo, la conversión a algoritmo no recursivo es simplemente escribir las instrucciones apropiadas para que el computador ejecute lo mismo que hacemos al hacer el seguimiento. Los pasos a seguir son:

1. Definir una pila vacía.
2. Asignar label (rótulo) a la primera instrucción ejecutable del subprograma.
3. Definir dirección de retorno, dependiendo si es una función o un procedimiento.
4. Asignar label (rótulo) a las instrucciones definidas como direcciones de retorno en el paso 3.
5. Remover llamadas recursivas. Por cada llamada recursiva:
   5.1 Guardar en la pila:
   - Dirección de retorno.
   - Parámetros por valor del subprograma.
   - Variables locales del subprograma.

   5.2 Asignar nuevos valores a los parámetros del subprograma.
   5.3 Transferir el control a la instrucción cuyo label se definió en el paso 2.
6. Antes de cualquier EXIT o END del subprograma codificar instrucciones que hagan lo siguiente:
   Controlar que la pila no esté vacía (IF TOPE > 0)

   Si la pila no está vacía desapilar los datos correspondientes a una llamada recursiva: Parámetros, variables locales y dirección de retorno, y transferir el control a la instrucción rotulada con la dirección de retorno extraída de la pila.

El procedimiento descrito anteriormente es completamente válido para PROCEDIMIENTOS, si se trata de una función hay que hacer unas pequeñas modificaciones, las cuales trataremos cuando hagamos un ejemplo con una de ellas.

Ilustraremos los pasos definidos con el procedimiento MOVER_TORRE.

Su versión recursiva es:

```
1  sub_programa mover_torre(n, o, a, d)
2      if n > 0 then
3          mover_torre(n-1,o,d,a)
4          write(o,'—>',d)
5          mover_torre(n-1,a,o,d)
```

```
6       end(if)
7  fin(mover_torre)
```

la versión no recursiva es:

```
1  sub_programa mover_torre(n, o, a, d)
2       dim pila(100)
3       tope = 0
4  l0:  if  n > 0  then
5            pila(tope + 1) = n
6            pila(tope + 2) = o
7            pila(tope + 3) = a
8            pila(tope + 4) = d
9            pila(tope + 5) = 'l1'
10           tope = tope + 5
11           n = n - 1
12           o = o
13           a = d
14           d = pila(tope - 2)
15           goto l0
16 l1:       write(o,'→',d)
17           pila(tope + 1) = n
18           pila(tope + 2) = o
19           pila(tope + 3) = a
20           pila(tope + 4) = d
21           pila(tope + 5) = 'l2'
22           tope = tope + 5
23           n = n - 1
24           o = a
25           a = pila(tope - 3)
26           d = d
27           goto l0
28 l2: end(if)
29      if tope > 0 then
30           tope = tope - 5
31           n = pila(tope + 1)
32           o = pila(tope + 2)
33           a = pila(tope + 3)
34           d = pila(tope + 4)
35           dr = pila(tope + 5)
36           goto dr
37      end(if)
38 fin(mover_torre)
```

Las líneas 2 y 3 son la definición de la pila vacía.

En la línea 4 se le asigna rótulo (label) L0 a la primera instrucción ejecutable del subprograma.

Las direcciones de retorno serán L1 y L2, con las cuales rotularemos las instrucciones siguientes a las llamadas recursivas, las cuales son las instrucciones WRITE (16) y END (28) respectivamente.

Las líneas 5 a 15 son las que reemplazan la primera llamada recursiva.

En líneas 5 a 8 se apilan los parámetros del procedimiento.

En la línea 9 se apila la dirección de retorno L1, que es con la cual se rotula la instrucción WRITE, la cual es la instrucción siguiente a la primera llamada recursiva.

Líneas 11 a 14 asignan nuevos valores a los parámetros, y línea 15 transfiere el control al principio del procedimiento.

Línea 16 es simplemente la instrucción donde continúa el proceso cuando termina de resolver la primera llamada recursiva. Es la instrucción a la cual se le asigna rótulo (dirección de retorno).

Líneas 17 a 27 reemplazan la segunda llamada recursiva: 17 a 22 apilan los datos correspondientes a la llamada que se está interrumpiendo; 23 a 26 asignan los nuevos valores a los parámetros; línea 27 retorna el control al principio del procedimiento.

Línea 28 es la instrucción que es dirección de retorno correspondiente a la segunda llamada recursiva.

Líneas 29 a 37 son las instrucciones para desapilar los datos correspondientes a una llamada recursiva que fue interrumpida. El control se transfiere a L1 o a L2 dependiendo del dato que se haya sacado de la pila.

Hasta aquí es el proceso, digamos mecánico, que se sigue para convertir un procedimiento recursivo en uno no recursivo. En esta primera versión, el algoritmo obtenido tiene instrucciones redundantes y una o varias instrucciones GOTO. El paso siguiente es hacer un análisis lógico del algoritmo para eliminar las instrucciones redundantes y los GOTO.

En nuestro algoritmo, las instrucciones 12 y 26 son, a todas luces, instrucciones que sobran. Veamos ahora, otras instrucciones que no es tan obvio que sobren, pero que sobran. Cuando pregunta si TOPE es mayor que cero y resulta verdadera la condición, desapila los datos correspondientes a una llamada recursiva con su correspondiente dirección de retorno y regresa a L1 o a L2 dependiendo de lo que hubiera desapilado. Cuando retorna a L2, comienza a ejecutar a partir del END(if) de la instrucción 28, la instrucción siguiente es preguntar de nuevo si hay datos correspondientes a una llamada interrumpida para volver a desapilar, por consiguiente los datos correspondientes a una llamada que se interrumpe, con dirección de retorno L2 no tienen utilización alguna y guardarlos no tiene sentido. De lo anterior, deducimos que las instrucciones para apilar estos datos sobran, es decir, las instrucciones 17 a 22. Ahora bien, si no vamos a guardar estos datos, la única dirección de retorno que quedará en la pila será L1, o sea que no habrá necesidad de almacenarla tampoco en la pila y la instrucción 9 también sobrará y la instrucción 36 se reemplaza por una instrucción que sea GOTO L1. Con estas modificaciones el algoritmo quedará así:

```
1  sub_programa mover_torre(n, o, a, d)
2       dim pila(100)
3       tope = 0
4  l0:  if n > 0 then
5            pila(tope + 1) = n
6            pila(tope + 2) = o
7            pila(tope + 3) = a
8            pila(tope + 4) = d
9            tope = tope + 4
10           n = n - 1
11           a = d
12           d = pila(tope - 1)
13           goto l0
14 l1:       write(o,'→',d)
15           n = n - 1
16           o = a
17           a = pila(tope + 2)
18           goto l0
19      end(if)
20      if tope > 0 then
21           tope = tope - 4
22           n = pila(tope + 1)
23           o = pila(tope + 2)
24           a = pila(tope + 3)
25           d = pila(tope + 4)
26           goto l1
```

```
27      end(if)
28 fin(mover_torre)
```

El paso siguiente es eliminar los GOTO:

Cuando la condición de la instrucción 4 es verdadera ejecuta una serie de instrucciones y regresa siempre a chequear de nuevo la condición mediante la instrucción 13. Este grupo de instrucciones es simplemente un ciclo WHILE: Mientras N sea mayor que cero.

La única forma que ejecute a partir de la instrucción 14 es que tope hubiera sido mayor que cero, ya que es allí donde se transfiere el control a la instrucción 14, por consiguiente, desde la instrucción 14 hasta la instrucción 18 pertenecen a la condición de la instrucción 20 y pueden ser trasladadas allí a reemplazar la instrucción 26. Nuestro algoritmo queda así:

```
1  sub_programa mover_torre(n, o, a, d)
2       dim pila(100)
3       tope = 0
4  l0:  while n > 0 do
5            pila(tope + 1) = n
6            pila(tope + 2) = o
7            pila(tope + 3) = a
8            pila(tope + 4) = d
9            tope = tope + 4
10           n = n - 1
11           a = d
12           d = pila(tope - 1)
13      end(while)
14      if tope > 0 then
15           tope = tope - 4
16           n = pila(tope + 1)
17           o = pila(tope + 2)
18           a = pila(tope + 3)
19           d = pila(tope + 4)
20           write(o,'→',d)
21           n = n - 1
22           o = a
23           a = pila(tope + 2)
24           goto l0
25      end(if)
26 fin(mover_torre)
```

Falta por eliminar el GOTO de la instrucción 24. La única forma de que esta instrucción se ejecute es que TOPE sea mayor que cero. La manera de eliminarlo es utilizando un ciclo, controlado por la condición de que TOPE sea mayor que cero. Usamos la instrucción REPEAT debido a que con la instrucción WHILE no entraría al ciclo ya que el valor inicial de TOPE es cero. Nuevamente, de la instrucción 16 a la 23 hay instrucciones que sobran: las instrucciones 16 y 21 son una doble asignación a la variable N, las instrucciones 17 y 22 son doble asignación a la variable O y las 18 y 23 a la variable A. Reagrupando estas instrucciones y eliminando el GOTO nuestro algoritmo queda así:

```
1  sub_programa mover_torre(n, o, a, d)
2      dim pila(100)
3      tope = 0
4      repeat
5          while n > 0 do
6              pila(tope + 1) = n
7              pila(tope + 2) = o
8              pila(tope + 3) = a
9              pila(tope + 4) = d
10             tope = tope + 4
11             n = n - 1
12             a = d
13             d = pila(tope - 1)
14         end(while)
15         if tope > 0 then
16             tope = tope - 4
17             n = pila(tope + 1) - 1
18             o = pila(tope + 3)
19             a = pila(tope + 2)
20             d = pila(tope + 4)
21             write(a,'→',d)
22         end(if)
23     until tope = 0  and  n = 0
24 end(sub_programa)
```

10.4 PERMUTACIONES

El problema de las permutaciones es uno de los clásicos en análisis combinatorio. Veamos cómo analizar y escribir un algoritmo que efectúe dicha labor.

Consideraremos que los datos a permutar se hallan en un vector de **n** elementos llamado **V**. En nuestro ejemplo **n** = 11.

	1	2	3	4	5	6	7	8	9	10	11
V	a	b	c	d	e	f	g	h	i	j	k

Si deseo producir las permutaciones de **n** elementos, dejo fijo el primer dato (el de la posición 1) y produzco las permutaciones de los **n – 1** datos restantes (de la posición 2 hasta la **n**, (n=11)); cuando haya terminado con estas permutaciones intercambio el dato de la posición 1 (que estaba fija) con el dato de la posición 2 y vuelvo a producir las permutaciones de los datos desde la posición 2 hasta la posición **n** (n=11); terminado esto, intercambio el dato de la posición 1 con el dato de la posición 3 y nuevamente repito permutaciones desde la posición 2 hasta la 11. En otras palabras, tengo que intercambiar el dato de la posición 1 con cada uno de los demás datos y por cada intercambio debo hacer las permutaciones de los **n – 1** datos restantes, de la posición 2 a la 11.

Para hacer las permutaciones de los datos desde la posición 2 hasta la posición **n** (n=11) el proceso es análogo: dejo fijo el dato de la posición 2 y produzco las permutaciones de los datos desde la posición 3 hasta la posición **n** (n=11); terminadas estas permutaciones intercambio el dato de la posición 2 con el dato de la posición 3 y nuevamente produzco las permutaciones de los datos desde la posición 3 hasta la posición **n**. O sea, el dato de la posición 2 lo debo intercambiar con todos los datos desde la posición 3 hasta la posición **n** y por cada intercambio producir las permutaciones desde la posición 3 hasta la posición **n**.

Generalizando el anterior proceso, debemos manejar una variable que indique a partir de qué posición es que se desean hacer permutaciones, llamemos **i** dicha variable, y otra variable que indique hasta qué posición se desean hacer las permutaciones, llamemos **n** esta otra variable.

El dato de la posición **i** debemos intercambiarlo con todos los datos desde la posición **i+1** hasta la posición **n** y por cada intercambio debemos hacer permutaciones (llamada recursiva) desde la posición **i+1** hasta la posición **n**.

El algoritmo que ejecuta dicho proceso es el siguiente:

```
1  sub_programa permuta(v, n, i)
2     if i = n then
3        write(v)
4     else
5        for k = i to n do
6           intercambie(v(i) con v(k))
```

```
7          permuta(v, n, i+1)
8 l1→ end(for)
9     end(if)
10 fin(permuta)
```

Sólo haremos una parte del seguimiento, la que consideramos suficiente para entender cómo funciona dicho algoritmo. Consideremos el siguiente vector V, con n=3:

1	2	3
a	b	C

El programa llamante del procedimiento PERMUTA es:

```
     — — — —
     — — — —
     PERMUTA(V, 3, 1)
L0 → — — — —
     — — — —
```

Las direcciones de retorno las rotularemos L0 y L1 respectivamente, las cuales son las instrucciones siguientes a las instrucciones que llaman a PERMUTA, ya que se trata de un procedimiento. La pila para manejar los datos correspondientes a llamadas interrumpidas contendrá la información perteneciente a los parámetros (**V, n, i**), la dirección de retorno y la variable local **k**.

Cuando se ejecuta la llamada a PERMUTA en el programa llamante el estado de la pila es el siguiente:

L0	a	b	c	3	1	
DR		**V**		**n**	**i**	**k**

que significa que se va a ejecutar el procedimiento PERMUTA con los datos correspondientes de **V, n** e **i** y que cuando termine regresará a la instrucción llamada L0 a continuar ejecutando el programa llamante.

La condición de la instrucción 2 es falsa, por consiguiente continúa ejecutando a partir de la instrucción 5. Esta instrucción es el comienzo de un ciclo FOR, el

cual hará que la variable **k** tenga 3 valores: 1, 2 y 3, y comenzará ejecutando con el valor de **k=1.** Esta situación la representaremos en la pila de la siguiente forma:

L0	a	b	c	3	1	1	2	3
DR		**V**		**n**	**i**		**k**	

representando que vamos a ejecutar el ciclo FOR con la variable **k = 1.**

Las instrucciones del ciclo FOR son: intercambiar el contenido de V(I) con V(K), lo cual deja el vector sin modificación debido a que **i** y **k** tienen el mismo valor; la instrucción siguiente es llamada recursiva al procedimiento PERMUTA. El estado de la pila será el siguiente:

L1	a	b	c	3	2			
L0	a	b	c	3	1	1	2	3
DR		**V**		**n**	**i**		**k**	

indicando que vamos a ejecutar dicho procedimiento con los valores de **V, n** e **i** que hay en el tope y con dirección de retorno a la instrucción L1 para continuar ejecutando el ciclo FOR de la llamada interrumpida.

La nueva ejecución ocasiona que la condición de la instrucción 2 sea falsa, por tanto, la ejecución continúa a partir de la instrucción 5, la cual inicia el ciclo FOR con **k** variando desde 2 hasta 3 y su valor inicial es 2. El estado de la pila es:

L1	a	b	c	3	2	2	3	
L0	a	b	c	3	1	1	2	3
DR		**V**		**n**	**i**		**K**	

indicando que comenzaremos la ejecución del ciclo FOR con **k=2.**

La primera instrucción del ciclo intercambia **V(i)** con **V(k)**, dejando el vector inalterado (**k** e **i** son iguales) y encuentra una nueva llamada recursiva al procedimiento PERMUTA, que en la pila representamos así:

L1	a	b	c	3	3			
L1	a	b	c	3	2	**2**	3	
L0	a	b	c	3	1	**1**	2	3
DR		**V**		**n**	**i**		**k**	

indicando que ejecutaremos PERMUTA con los valores de **V**, **n** e **i** que hay en el tope, con dirección de retorno a la instrucción rotulada L1 para continuar la ejecución del ciclo que se interrumpió con la variable **k** valiendo 2.

La ejecución de PERMUTA con estos nuevos valores de los parámetros ocasiona que la condición de la instrucción 2 sea verdadera, ejecutando por consiguiente la instrucción 3, la cual imprime el vector **V**, produciendo la siguiente impresión: **abc**.

Continúa con la instrucción siguiente al END(if), la cual es el fin del procedimiento, indicando que ha terminado la ejecución del procedimiento PERMUTA, por tanto desapila los datos que hay en el tope de la pila y retorna a la instrucción rotulada L1 a continuar el ciclo FOR de la llamada que había sido interrumpida. El estado de la pila es el siguiente:

L1	a	b	c	3	2	**2**	3	
L0	a	b	c	3	1	**1**	2	3
DR		**V**		**n**	**i**		**k**	

La instrucción L1 es el fin del ciclo FOR, lo que significa que ha terminado de ejecutar el ciclo para **k=2** y que debe repetir de nuevo el ciclo con **k=3**. Representaremos esta situación en la pila así:

L1	a	b	c	3	2	2	**3**	
L0	a	b	c	3	1	**1**	2	3
DR		**V**		**N**	**I**		**K**	

indicando que hemos ejecutado el ciclo FOR con **k=2** y que continuamos la ejecución del ciclo con **k=3**.

Las instrucciones del ciclo son: intercambiar **V(i)** con **V(k)**, por tanto intercambia **V(2)** con **V(3)** y el estado de la pila es el siguiente:

L1	a	c	b	3	2	**2**	**3**	
L0	a	b	c	3	1	**1**	2	3
DR		**V**		**n**	**i**		**k**	

Luego llama nuevamente a PERMUTA y la pila queda así:

L1	a	c	b	3	3			
L1	a	c	b	3	2	**2**	**3**	
L0	a	b	c	3	1	**1**	2	3
DR		**V**		**n**	**i**		**K**	

indicando que ejecutará el procedimiento con los datos que hay en el tope y que retornará a L1 para continuar la ejecución del ciclo FOR con **k=3**.

Esta nueva ejecución ocasiona que la condición de la instrucción 2 sea verdadera, por tanto imprime el contenido del vector **V**, produciendo la siguiente impresión: **acb**.

Continúa con la instrucción siguiente al END(if), la cual es el fin del procedimiento, por tanto desapila los datos que hay en el tope y regresa a la instrucción llamada L1 para continuar la ejecución del ciclo FOR con **k=3**. El estado de la pila es el siguiente:

L1	a	c	b	3	2	**2**	**3**	
L0	a	b	c	3	1	**1**	2	3
DR		**V**		**n**	**i**		**k**	

La instrucción L1 es el fin del FOR, lo que significa que ha terminado de ejecutar el ciclo FOR para **k=3**, pero como éste es el último valor que había de tomar la **k**, el ciclo FOR ha terminado para esta llamada y continúa con el END(if) y luego el fin del procedimiento y ha terminado de ejecutar esta llamada. Entonces desapila y regresa a L1 a continuar ejecutando el ciclo de la llamada que había interrumpido. El estado de la pila ahora es:

L0	a	b	c	3	1	1	2	3
DR		**V**		**n**	**i**		**K**	

La instrucción L1 es el fin del FOR, o sea que ha terminado para **k=1** y debe repetir las instrucciones del ciclo con **k=2**. Representamos esto en la pila así:

L0	a	b	c	3	1	1	**2**	3
DR		**V**		**n**	**i**		**k**	

indicando que ejecutaremos de nuevo las instrucciones del ciclo FOR con **k=2**.

Las instrucciones del ciclo FOR son: intercambiar el contenido de **V(i)** con **V(k)** (intercambia V(1) con V(2)) y llamada recursiva a PERMUTA con el nuevo estado del vector **V** y **n=3** e **i=2**. el estado de la pila es:

L1	b	a	c	3	2			
L0	a	b	c	3	1	1	**2**	3
DR		**V**		**n**	**i**		**k**	

A partir de este punto se presenta una situación análoga a la que hemos visto anteriormente, en la cual se inicia de nuevo el ciclo FOR con valores para K de 2 y 3, produciendo las impresiones **bac** y **bca** respectivamente.

Al retornar a L1, habrá terminado el ciclo para **k=2** y lo repetirá de nuevo para **k=3**, lo cual producirá las impresiones **cab** y **cba**. Dejamos al estudiante que compruebe dichos resultados continuando con el seguimiento.

Ahora, a partir del algoritmo PERMUTA desarrollado, encontraremos la versión NO recursiva aplicando los pasos definidos con anterioridad.

La versión recursiva es:

```
1  sub_programa permuta(v, n, i)
2     if  i = n  then
3        write(v)
4     else
5        for  k = i to n  do
6           intercambie(v(i) con v(k))
7           permuta(v, n, i+1)
8        end(for)
9     end(if)
10 fin(permuta)
```

La versión NO recursiva aplicando los pasos dados es:

```
1.     sub_programa permuta(v, n, i)
2.          dim pila(100)
3.          tope = 0
4.  l0:     if  n = i  then
5.                write(v)
6.          else
7.                for  k = i to n  do
8.                     intercambie(v(i) con v(k)
9.                     pila(tope+1) = v
10.                    pila(tope+2) = n
11.                    pila(tope+3) = i
12.                    pila(tope+4) = k
13.                    pila(tope+5) = 'l1'
14.                    tope = tope + 5
15.                    v = v
16.                    n = n
17.                    i = i + 1
18.                    goto l0
19. l1:           end(for)
20.         end(if)
21.         if tope > 0 then
22.               tope = tope - 5
```

```
23.             v = pila(tope+1)
24.             n = pila(tope+2)
25.             i = pila(tope+3)
26.             k = pila(tope+4)
27.             dr = pila(tope+5)
28.             goto dr
29.         end(if)
30.   fin(permuta)
```

Esta es la primera versión resultante. Como dijimos anteriormente un buen análisis del algoritmo permite eliminar instrucciones redundantes y los GOTO:

Es obvio que las instrucciones 15 y 16 sobran; sólo se guarda una dirección de retorno, (L1), luego las instrucciones 13 y 27 también sobran, basta cambiar la instrucción 28 por GOTO L1; la variable N conserva el mismo valor a través de todas las llamadas, por tanto es tontería guardarla en la pila, o sea que las instrucciones 10 y 24 también sobran. Eliminando estas instrucciones nuestro algoritmo queda así:

```
1.    sub_programa permuta(v, n, i)
2.          dim pila(100)
3.          tope = 0
4.  l0:     if n = i then
5.                write(v)
6.          else
7.                for k = i  to  n  do
8.                      intercambie(v(i) con v(k)
9.                      pila(tope+1) = v
10.                     pila(tope+2) = i
11.                     pila(tope+3) = k
12.                     tope = tope + 3
13.                     i = i + 1
14.                     goto l0
15. l1:           end(for)
16.         end(if)
17.         if tope > 0 then
18.               tope = tope - 3
19.               v = pila(tope+1)
20.               i = pila(tope+2)
21.               k = pila(tope+3)
22.               goto l1
23.         end(if)
24.   end(sub_programa)
```

La eliminación de los GOTO no es tan sencilla en este caso ya que cada vez que termina una llamada recursiva debe retornar a una instrucción END(for), cuya función es incrementar la variable controladora del ciclo en uno y si es menor o igual que el valor final de dicha variable, repite las instrucciones del ciclo. La instrucción FOR propiamente, lo que hace es asignar un valor inicial a la variable controladora del ciclo y controlar que sea menor o igual que el valor límite definido para entrar a ejecutar las instrucciones del ciclo.

Para poder eliminar los GOTO, nuestro primer paso será expresar el ciclo FOR con sus instrucciones elementales. Nuestro algoritmo queda así:

```
1.    sub_programa permuta(v, n, i)
2.        dim pila(100)
3.        tope = 0
4.  I0:   if n = i then
5.            write(v)
6.        else
7.            k = i
8   I2:       if k <= n then
9.                intercambie(v(i) con v(k)
10.               pila(tope+1) = v
11.               pila(tope+2) = i
12.               pila(tope+3) = k
13.               tope = tope + 3
14.               i = i + 1
15.               goto I0
16. I1:           k = k + 1
17                goto I2
18            end(if)
19.       end(if)
20.       if tope > 0 then
21.           tope = tope - 3
22.           v = pila(tope+1)
23.           i = pila(tope+2)
24.           k = pila(tope+3)
25.           goto I1
26.       end(if)
27.   fin(permuta)
```

Teniendo el algoritmo, con el ciclo que contiene la llamada recursiva, expresado en sus instrucciones elementales, hay una técnica con la cual, identificando la instrucción o grupo de instrucciones que se ejecutan secuencialmente como un bloque y utilizando una variable adicional, llamémosla ESTADO, se puede plantear

un ciclo con una instrucción CASE que de acuerdo al valor de ESTADO ejecute algún grupo de instrucciones y asigne a ESTADO el nuevo valor de acuerdo al conjunto de instrucciones con que continúe la ejecución.

En nuestro algoritmo los bloques de instrucciones secuenciales, «autónomos», que se identifican son los siguientes: instrucciones 4 a 7, las cuales ejecutaremos cuando ESTADO=0; instrucciones 8 a 14, las cuales ejecutaremos cuando ESTADO=1; instrucciones 20 a 24 junto con la instrucción 16, (ésta última sólo se ejecuta cuando se hayan ejecutado las instrucciones 20 a 24), las cuales ejecutaremos cuando ESTADO=2.

En cada uno de estos casos codificaremos dichas instrucciones y le asignaremos a la variable ESTADO un nuevo valor dependiendo del bloque de instrucciones que se deban ejecutar a continuación de acuerdo a la lógica del algoritmo.

Inicialmente asignaremos 0 a la variable ESTADO, y terminaremos el ciclo cuando ESTADO=3. Nuestro algoritmo quedará así:

Con esta técnica, cualquier algoritmo, por complicada maraña de GOTO's que tenga, siempre se podrá convertir a un algoritmo sin GOTO's.

Nuestro procedimiento PERMUTA sin GOTOS' es el siguiente:

```
sub_programa permuta(v, n, i)
    dim pila(100)
    tope = 0
    estado = 0
    while estado <> 3 do
        casos de estado
            0:  if n = i then
                    write(v)
                    estado = 2
                else
                    k = i
                    estado = 1
                end(if)
            1:  if k <= n then
                    intercambie(v(i) con v(k))
                    pila(tope+1) = v
                    pila(tope+2) = i
                    pila(tope+3) = k
                    tope = tope + 3
```

```
                    i = i + 1
                    estado = 0
                else
                    estado = 2
                end(if)
            2:  if tope > 0 then
                    tope = tope - 3
                    v = pila(tope+1)
                    i = pila(tope+2)
                    k = pila(tope+3) + 1
                    estado = 1
                else
                    estado = 3
                end(if)
        fin(casos)
    end(while)
fin(permuta)
```

10.5 CONVERSIÓN DE UNA FUNCIÓN RECURSIVA EN NO RECURSIVA

La conversión de una función recursiva a no recursiva difiere de la conversión de un procedimiento en algunos pequeños detalles: el primero es que el valor de la función se llevará siempre en el tope de la pila; el segundo es que la dirección de retorno es la misma instrucción que contiene la llamada recursiva, y el tercero es que la llamada recursiva se reemplaza siempre por PILA(TOPE) y la instrucción siguiente a la que contiene la llamada recursiva será siempre decrementar tope en 1. Ilustraremos el mecanismo convirtiendo la función FACT vista anteriormente a su versión no recursiva. El algoritmo recursivo es:

```
1    function fact(n)
2        if n = 0 then
3            fact = 1
4        else
5            fact = n * fact(n-1)
6        end(if)
7    fin(fact)
```

la versión no recursiva será:

```
1    function fact(n)
2        dim pila(100)
3        tope = 0
```

```
4  l0:      if  n = 0  then
5                fact = 1
6           else
7                pila(tope+1) = n
8                pila(tope+2) = 'l1'
9                tope = tope + 2
10               n = n - 1
11               goto l0
12 l1:           fact = n * pila(tope)
13               tope = tope - 1
14          end(if)
15          if  tope > 0  then
16               tope = tope - 1
17               n = pila(tope)
18               dr = pila(tope+1)
19               pila(tope) = fact
20               goto dr
21          end(if)
22     fin(fact)
```

La eliminación de instrucciones redundantes es sencilla. Como sólo hay una dirección de retorno no es necesario guardar en la pila dicha dirección, por tanto las instrucciones 8 y 18 sobran y la instrucción 20 se reemplaza por la instrucción GOTO L1.

Las instrucciones 12 y 13 sólo se ejecutan cuando TOPE = 0, por tanto se pueden trasladar antes de la instrucción 20 y se traslada el rótulo L1 hacia la instrucción IF TOPE > 0. Nuestro algoritmo quedará así:

```
1     function fact(n)
2          dim pila(100)
3          tope = 0
4  l0:     if n = 0 then
5               fact = 1
6          else
7               tope = tope + 1
8               pila(tope) = n
9               n = n - 1
10              goto l0
11         end(if)
12 l1:     if tope > 0 then
13              n = pila(tope)
14              pila(tope) = fact
```

```
15              fact = n * pila(tope)
16              tope = tope - 1
17              goto l1
18          end(if)
19  fin(fact)
```

Para eliminar el GOTO L0 vemos que las instrucciones 7 a 10 sólo se ejecutan cuando N es mayor que cero y se regresa a chequear de nuevo el valor de N, lo que configura un ciclo mientras. Las instrucciones 12 a 17 también configuran un ciclo mientras. Nuestro algoritmo queda así:

```
1     function fact(n)
2           dim pila(100)
3           tope = 0
4           while n > 0 do
5                 tope = tope + 1
6                 pila(tope) = n
7                 n = n - 1
8           end(while)
9           fact = 1
10          while tope > 0 do
11                n = pila(tope)
12                pila(tope) = fact
13                fact = n * pila(tope)
14                tope = tope - 1
15          end(while)
16  end(function)
```

Y este algoritmo muestra el típico funcionamiento de la recursión: guardar en una pila los datos correspondientes a llamadas interrumpidas y luego resolver para dichos datos.

EJERCICIOS PROPUESTOS

1. Escriba una función recursiva que calcule e imprima 2 utilizando únicamente la operación de suma.

2. Escriba un algoritmo recursivo que imprima las combinaciones de **n** elementos tomados de a **r**.

3. Escriba una función recursiva que evalúe el determinante asociado a una matriz cuadrada.

4. Escriba un algoritmo recursivo que imprima las 2^n posibles combinaciones de **n** variables lógicas.

5. El conjunto potencia, de un conjunto dado, se define como todos los subconjuntos que se pueden conformar a partir de los elementos de un conjunto dado. Escriba un algoritmo recursivo que imprima el conjunto potencia de un conjunto dado.

6. La información correspondiente a los padres de un mico se guardan en una tripleta de la siguiente forma: el primer dato corresponde al código del mico, el segundo dato al código del papá del mico y el tercer dato al código de la mamá del mico. Dada una matriz de tripletas, escriba un algoritmo recursivo que imprima el árbol genealógico de un mico dado.

7. La función de Ackerman se define así:
   f(m,n) = n+1 si m=0
   f(m,n) = f(m-1,1) si n=0
   f(m,n) = f(m-1,f(m,n-1)) en cualquier otro caso.

   Escriba un algoritmo recursivo para evaluar dicha función.

8. Escriba una función recursiva que calcule la suma de cualquier cantidad de elementos de un vector dado.

9. Escriba versiones NO recursivas sin GOTO'S de todos los algoritmos que ha escrito anteriormente.

10. Haga seguimiento a los siguientes algoritmos recursivos y determine la diferencia entre ellos.

```
sub_programa recorre_lista(l)
        p = l
        if  p <> 0  then
                write(dato(p))
                recorre_lista(liga(p))
        end(if)
fin(recorre_lista)
```

```
sub_programa recorre_lista(l)
        p = l
        if  p <> 0  then
                recorre_lista(liga(p))
        write(dato(p))
        end(if)
fin(recorre_lista)
```

11

LISTAS GENERALIZADAS

11.1 DEFINICIÓN

Una lista generalizada es un conjunto finito de **n** elementos **(n >= 0)** cada uno de los cuales es un átomo u otra lista.

La lista generalizada es una estructura recursiva por definición.

Por ejemplo: L = (a, (b, c), d, (e, (f, g)), h) {1}

Los átomos se presentan separados por comas, y cuando un elemento sea otra lista, ésta irá entre paréntesis.

Adoptemos como notación que las letras mayúsculas designarán listas y que las letras minúsculas designarán átomos. Por ejemplo:

A = (a, b, c, d)
B = (a, A, f, A, g)
C = (A, B, x, A)

Si expandimos las listas B y C tendremos:

B = (a, (a, b, c, d), f, (a, b, c, d), g)
C = ((a, b, c, d), (a, (a, b, c, d), f, (a, b, c, d), g), x, (a, b, c, d))

11.2 REPRESENTACIÓN DE LISTAS GENERALIZADAS

La conformación del registro usa 3 campos:

SW	Dato	Liga

$$SW = \begin{cases} 0: \text{En el campo de dato hay un átomo} \\ 1: \text{El campo dato es un apuntador hacia una sublista.} \end{cases}$$

Cada elemento de la lista generalizada utiliza un registro (o nodo).

Representemos la lista **L** presentada en {1} en la figura 11.1:

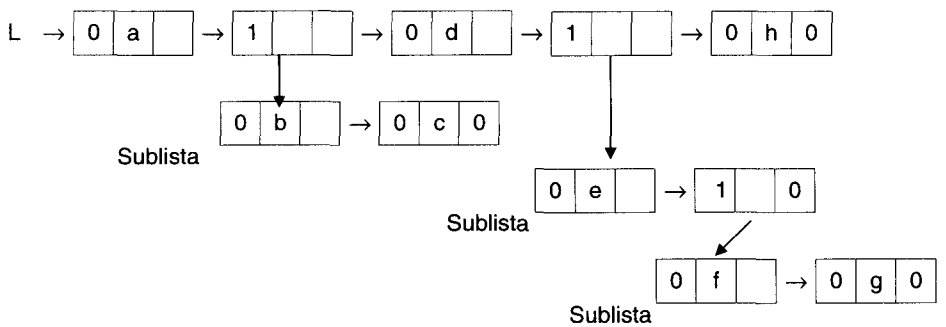


Figura 11.1

11.3 CONSTRUCCIÓN DE LA LISTA LIGADA QUE REPRESENTA UNA LISTA GENERALIZADA

Para construir la lista ligada que representa una lista generalizada lo primero que debemos hacer es definir la forma como entrarán los datos al programa de construcción. La entrada será una hilera de paréntesis izquierdos, átomos, comas y paréntesis derechos, como en {1}.

Supondremos que la hilera que representa la lista generalizada está bien construida, es decir comienza con un paréntesis izquierdo y termina con un paréntesis derecho.

Nuestro algoritmo recibe como parámetro de entrada una hilera **s**, en la cual está la hilera de paréntesis izquierdos, átomos, comas y paréntesis derechos.

Nuestro algoritmo es:

```
function conslg(s)
1.    define pila(100)
2.    tope = 0
3.    new(x)
```

```
4.    L = x
5.    ultimo = x
6.    n = longitud(s)
7.    for i = 2 to n - 1 do
8.          casos de s(i)
9.                :átomo:
10.                     sw(ultimo) = 0
11.                     dato(ultimo) = s(i)
12.               :',':
13.                     new(x)
14.                     liga(ultimo) = x
15.                     ultimo = x
16.               :'(':
17.                     tope = tope + 1
18.                     pila(tope) = ultimo
19.                     new(x)
20.                     sw(ultimo) = 1
21.                     dato(ultimo) = x
22.                     ultimo = x
23.               :')':
24.                     liga(ultimo) = 0
25.                     ultimo = pila(tope)
26.                     tope = tope - 1
27.         fin(casos)
28.   end(for)
29.   liga(ultimo) = 0
30.   return(L)
fin(conslg)
```

Instrucciones 1 y 2 definen una pila vacía, la cual se representa en un vector.

Instrucciones 3 a 5 inician la creación de la lista consiguiendo un registro, señalándolo como el primero con la variable **L**, y a la vez, como la forma de construir la lista será añadiendo registros siempre al final de la lista, identificamos este registro como el último, utilizando una variable llamada **ultimo**.

Instrucción 6 determina el número de caracteres de la hilera de entrada haciendo uso de la función **longitud**, la cual ejecuta esta tarea.

Instrucciones 7 a 28 conforman el ciclo básico del algoritmo. En la hilera de entrada, los únicos símbolos son: paréntesis izquierdos, átomos, comas y paréntesis derechos. Por consiguiente planteamos una estructura **caso**, y dependiendo de cuál sea el símbolo en la posición **i,** efectuamos las operaciones correspondientes.

Si **s(i)** es un átomo basta con configurar el último registro con el **sw** en cero y llevar el átomo **s(i)** al campo de dato (instrucciones 9 a 11). Si **s(i)** es una coma indica que a continuación vendrá otro átomo u otra lista, por consiguiente conseguimos un nuevo registro, lo conectamos con el último y decimos que éste es el nuevo último (instrucciones 12 a 15). Si **s(i)** es un paréntesis izquierdo significa que hay que construir una sublista, la cual dependerá del registro que llevamos de último; como esto es una interrupción a la construcción de una lista entonces guardamos la dirección del **último** en la pila, conseguimos un nuevo registro, el cual será el nuevo último, y lo conectamos con el último a través del campo de dato (instrucciones 16 a 22). Si **s(i)** es un paréntesis derecho significa que se ha terminado de construir una sublista, por consiguiente le asignamos 0 al campo de liga de **ultimo** y regresamos a continuar construyendo la lista cuya construcción se había interrumpido asignándole a **ultimo** la dirección del registro que tenemos en el tope de la pila (instrucciones 23 a 26).

Al salirnos del ciclo terminamos la construcción de la lista principal y le asignamos 0 al campo de liga del último registro y retornando el valor de **L** (instrucciones 29 y 30).

A continuación presento un algoritmo recursivo que efectúa exactamente la misma tarea: construir la lista ligada que representa una lista generalizada.

```
function conslg(s)
      if  j <= n  then
            new(x)
            if  s(j) = '('  then
                  sw(x) = 1
                  j = j + 1
                  dato(x) = conslg(s)
            else
                  sw(x) = 0
                  dato(x) = s(j)
                  j = j + 1
            end(if)
            if  s(j) = ','  then
                  j = j + 1
                  liga(x) = conslg(s)
            else
                  liga(x) = 0
                  j = j + 1
            end(if)
            return(x)
      end(if)
fin(conslg)
```

El anterior algoritmo recibe como parámetro la hilera de paréntesis izquierdos, átomos, comas y paréntesis derecho, la cual denominamos **s**.

Las variables **n** y **j** son variables globales.

La variable **n** contiene el número de caracteres de la hilera **s**.

La variable **j** es la variable con la cual recorremos la hilera **s**.

Dejo al estudiante la tarea de comprobar la correctitud de dicho algoritmo.

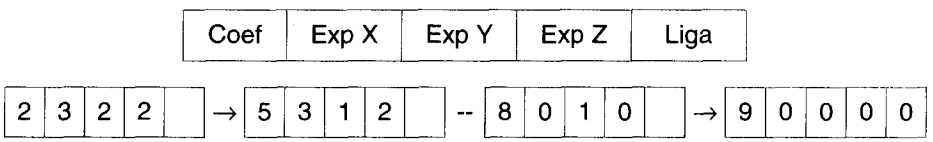
11.4 POLINOMIOS CON MÁS DE UNA VARIABLE

Como ejemplo de aplicación de lista generalizadas trataremos el manejo de polinomios en más de una variable.

Consideremos el siguiente polinomio:

$$2X^3y^2Z^2 + 5X^3YZ^2 + 7XY^2Z^2 + 5X^2YZ + 4XZ + 4X^3Y^2 + 8Y + 9 \quad \{2\}$$

Una forma de representar dicho polinomio sería como lista ligada. Para ello definimos un registro con la siguiente configuración:



Esta representación es ineficiente, en cuanto a consumo de memoria, para términos en los cuales el exponente de alguna variable sea cero. Es decir, para el término independiente del ejemplo tenemos tres campos de memoria cuyo valor es cero y ya hemos visto que esto se considera desperdicio de memoria.

Además esta representación es muy rígida en el sentido de que sólo manejará polinomios de tres variables. Si deseamos manejar polinomios en 5 variables, por ejemplo, debemos definir otra configuración de registro y elaborar el software correspondiente a esa configuración. En definitiva, nos llenaríamos de múltiples representaciones y de un software específico para cada representación, dependiendo del número de variables de los polinomios a trabajar.

El objetivo es definir una representación única que nos permita trabajar con cualquier polinomio, sin importar el número de variables que él tenga.

Si factorizamos el polinomio dado en {2} tendremos:

$$(2X^3Y^2 + 5X^3Y + 7XY^2)Z^2 + (5X^2Y + 4X)Z + (4X^3Y^2 + 8Y + 9)$$

el cual, perfectamente podemos decir que es un polinomio en la variable Z.

Si factorizamos los coeficientes de los términos en Z obtenemos lo siguiente:

$$((2Y^2 + 5Y)X^3 + (7Y^2)X)Z^2 + ((5Y)X^2 + (4)X)Z + ((4Y^2)X^3 + (8Y + 9))$$

El cual es un polinomio en Z cuyos coeficientes son polinomios en X, los cuales a su vez, tienen como coeficientes polinomios en Y. Como vemos, podemos perfectamente decir, que tenemos un polinomio en una sola variable, cuyos coeficientes son polinomios en una sola variable, etc. Si deseamos representarlo como lista generalizada debemos definir primero que todo la configuración del registro.

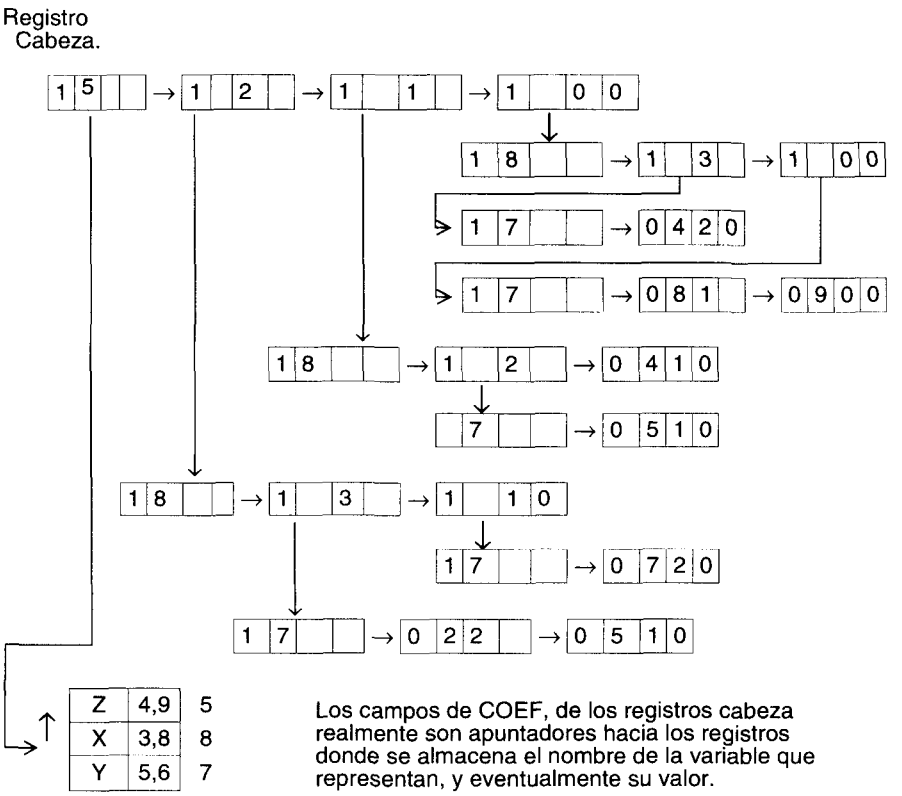
La configuración de registro es:

SW	Coef	Exp	Liga

Y puede manipularse en forma de lista generalizada

SW =
- 0: el campo COEF es un escalar
- 1: el campo COEF es un apuntador a la Lista que representa el polinomio que es coeficiente.

Cada polinomio se representa como una lista simplemente ligada con registro cabeza. Se usa un nodo por cada término del polinomio. Representemos el polinomio anterior factorizado. El registro cabeza tendrá SW = 1, y su campo de coeficiente será un apuntador.



EJERCICIOS PROPUESTOS

1. Elabore un algoritmo que copie una lista generalizada en otra.

2. Elabore un algoritmo que determine si dos listas generalizadas son iguales.

3. Elabore un algoritmo que evalúe un polinomio representado como lista generalizada.

4. Escriba un algoritmo recursivo que construya una lista generalizada dada la representación de paréntesis izquierdos, átomos, comas y paréntesis derechos.

5. Elabore un algoritmo que imprima una lista generalizada como una hilera de átomos, paréntesis izquierdos y paréntesis derechos dada la representación como lista ligada.

6. Elabore un algoritmo que reverse una lista generalizada y todas sus sublistas.

12

ÁRBOLES

12.1 DEFINICIÓN

Un árbol es un conjunto de **n** registros (**n > 0**, árbol vacío no está definido), tales que, hay un registro especial llamado RAIZ y los demás registros están particionados en conjuntos disjuntos, cada uno de los cuales es un árbol.

Un árbol es una estructura recursiva por definición, además cada registro que tenga hijos se podrá considerar como la raíz de un sub-árbol.

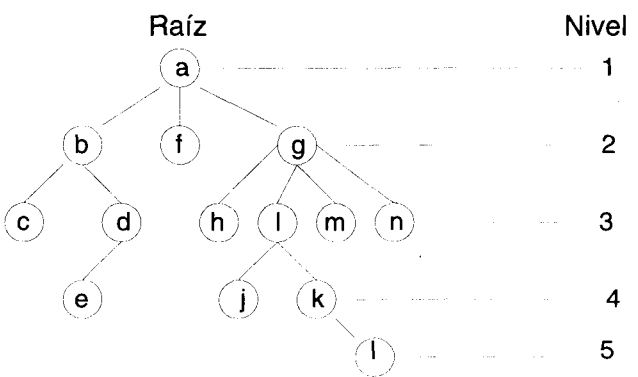


Figura 12.1

Las ramificaciones de cada nodo o registro suelen llamarse **hijos** y los nodos de los cuales salen las ramificaciones se llaman **padres**. Los registros que tienen un mismo **padre** se denominan **hermanos**.

12.2 TERMINOLOGÍA DE ÁRBOLES

Grado de un Registro. Es el número de ramificaciones que salen de ese registro. En el ejemplo de la figura se tiene: A tiene grado 3, B tiene grado 2, J tiene grado 0.

Grado de un Árbol. Es el máximo grado de cualquier registro del árbol. El árbol de la figura tiene grado 4.

Hojas. Son los registros con grado cero (registros que no tienen hijos). En el árbol del ejemplo las hojas son los registros que tienen los datos **f, c, h, m, n, e, j, l.**

A la raíz se le asigna el nivel 1, a sus hijos el nivel 2, a sus nietos el nivel 3, etc.

Para un registro que se halle en el nivel **i** su padre se halla en el nivel **i-1**, y sus hijos en el nivel **i+1**.

Altura de un Árbol. Es el máximo nivel de cualquier registro del árbol. En el ejemplo la altura es 5.

Ancestros de un Registro. Son todos los registros en la trayectoria desde la raíz hasta ese registro. Los ancestros de D son A, B y D.

12.3 REPRESENTACION DE ÁRBOLES

Una forma de hacerlo es usando Listas Ligadas. Recordemos que siempre que se usen Listas Ligadas debemos configurar el registro. En este caso debemos tener en cuenta el grado del árbol.

L 1	L 2	L 3	L 4	Dato

En caso de no utilizar alguna liga, su respectivo campo contendrá nil, o sea cero.

Sólo el registro G usa correctamente el registro de listas ligadas. Para los demás registros se presenta desperdicio de memoria. Sin embargo, esto no es muy grave, lo grave es que se desarrolla software para árboles de determinado grado solamente. Si tengo árboles con grado diferente, tendré que desarrollar software para cada grado, lo cual es impráctico, además de absurdo. Este problema se resuelve representando los árboles como listas generalizadas.

12.4 REPRESENTACIÓN DE ÁRBOLES COMO LISTAS GENERALIZADAS

La configuración del registro será de 3 campos:

SW	Dato	Liga

SW:
- 0 : En el campo de dato hay realmente un dato.
- 1 : En el campo de dato hay un apuntador hacia un subárbol.

Por cada raíz del árbol se tendrá una lista simplemente ligada cuyo primer registro representa la raíz del árbol y los demás registros representan los hijos de esa raíz. La Lista Generalizada que representa el árbol de la figura 12.1 es:

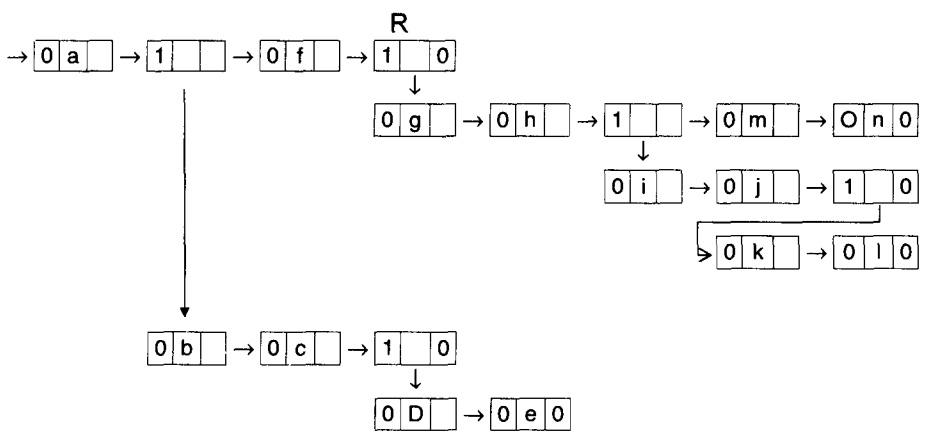


Figura 12.2

Para poder construir la lista generalizada de la figura 12.2, la forma como se entran los datos al computador es mediante una hilera de caracteres. La hilera de caracteres correspondiente al árbol anterior es:

(a(b(c, d(e)), f, g(h, i(j, k(l)), m, n)))

la cual, es muy semejante a la representación de lista generalizada vista anteriormente.

12.5 ÁRBOLES BINARIOS

Definición: un árbol binario es un conjunto de N registros (N >= 0), el cual, puede ser vacío o constar de una raíz y los demás registros particionados en dos conjuntos disjuntos, cada uno de los cuales es un árbol binario (Definición recursiva), que se conocen como sub-árbol izquierdo y sub-árbol derecho.

El árbol binario es una estructura recursiva por definición.

Es importante notar que el árbol binario vacío está definido.

Recuerde que en la definición de árboles en general el árbol vacío no estaba definido.

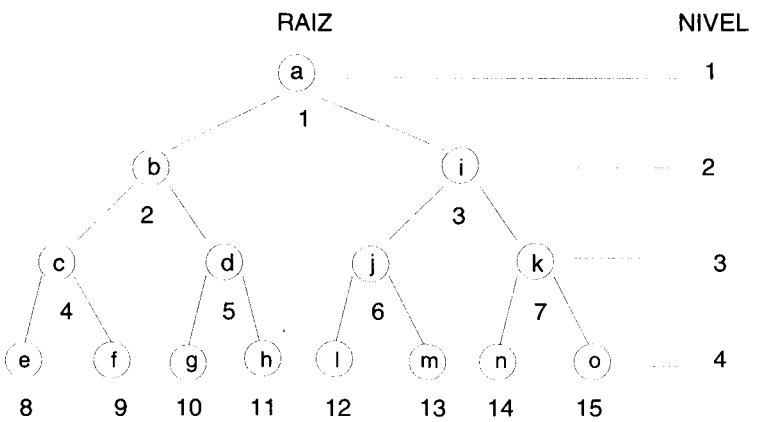


Figura 12.3

Toda la terminología definida en árboles es aplicable a árboles binarios.
Es obvio que el máximo grado de todo árbol binario es 2.

Propiedades del árbol binario:

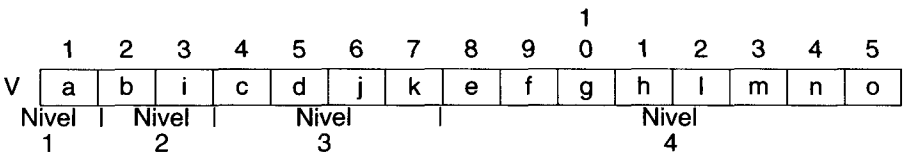
1. El máximo número de registros en un nivel **i** cualquiera es 2^{i-1}
2. Para un árbol binario de altura **K** el máximo número de registros es 2^{k-1}
   Arbol Lleno: es el árbol binario de altura **K** que tiene (2^{k-1}) registros.
3. Sea n_0 = Número de hojas del árbol (registros con grado 0).
   n_2 = Número de registros con grado 2.

Siempre se cumplirá: $n_0 = n_2 + 1$ excepto para el árbol vacío.

En el ejemplo de la figura 12.3:
$N_2 = 7$ y $N_0 = 8$

Representación de árboles binarios:

1. **En un vector:** el árbol de la figura 12.3 representado en un vector es:



Al nivel 1 le corresponde la posición 1 del vector; al nivel 2 le corresponden las siguientes dos posiciones del vector ya que el máximo número de registros en el nivel 2 es dos; al nivel tres le corresponden las siguientes 4 posiciones del vector, ya que el máximo número de registros del nivel 3 es cuatro, y así sucesivamente. Los registros en cada nivel se llenan de izquierda a derecha. A cada registro le corresponderá una posición fija del vector.

Si se tiene un árbol como el de la figura 12.4a su representación como vector se muestra en la figura 12.4b

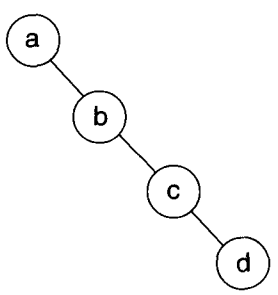


Figura 12.4a

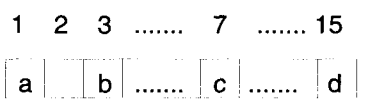


Figura 12.4b

Sólo se utilizan las posiciones 1, 3, 7 y 15.
Las demás posiciones del vector no se utilizan
Por tanto se consideran desperdicio de memoria.

Propiedades de la representación de un árbol binario en un vector:

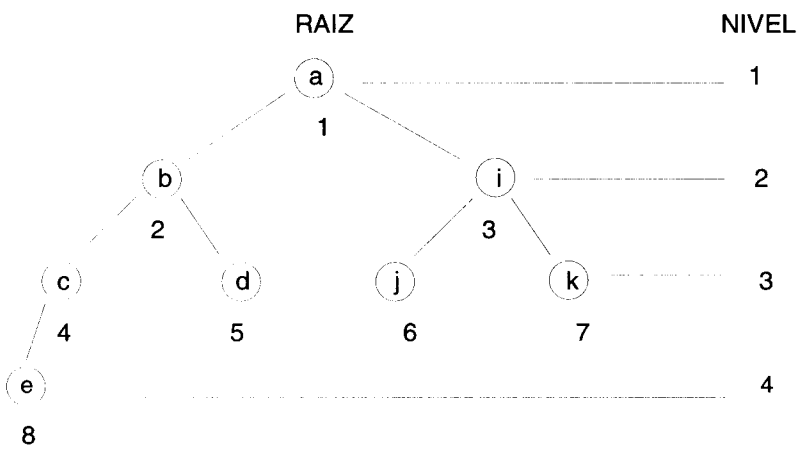
Para un registro en la posición **i** del vector:

1. Su padre se halla en la posición **i/2** del vector (i > 1). {división entera}
2. Su hijo izquierdo se halla en la posición **2*i**. (**2*i <= n, n**: número de registros del árbol).
3. Su hijo derecho se halla en la posición 2*i + 1. (**2*i + 1 <= n**)

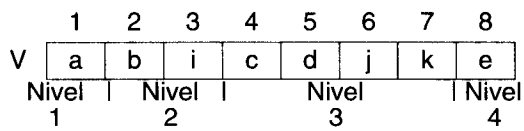
Árbol completo: es un árbol binario tal que al representarlo en un vector, utiliza efectivamente todas las posiciones del vector.

Todo Árbol Binario Lleno es Completo

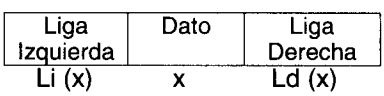
Lo contrario NO siempre se cumple, Por ejemplo: El siguiente es un árbol binario completo, aunque no sea lleno:



Al representarlo en un vector, se requieren 8 posiciones, y utilizamos efectivamente las ocho posiciones del vector.

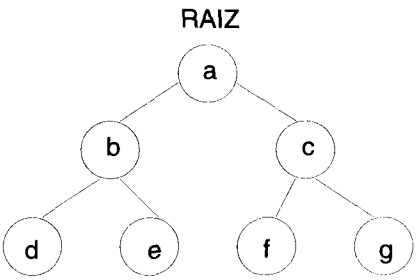


2. **Como listas ligadas:** siempre que se desee representar un objeto como lista ligada, lo primero que debemos hacer es definir la configuración del registro.

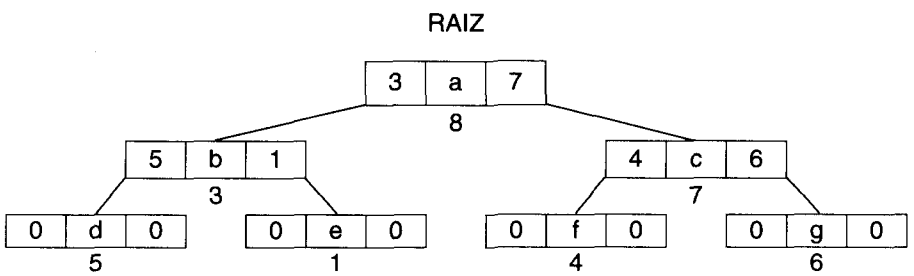


Li(x): Apunta hacia el registro que es un hijo izquierdo de x.
Ld(x): Apunta hacia el registro que es un hijo derecho de x.

Si tenemos el siguiente árbol binario:



Su representación como lista ligada será:



Con esta representación se usan los registros que efectivamente se necesitan.

Con esta representación podremos conocer fácil y eficientemente los hijos de cada registro, ya que los campos de liga izquierda y liga derecha apuntan hacia los registros que los representan. Si la aplicación que se desea implementar, manejando árboles, requiere permanentemente conocer el padre de un registro, entonces se incluye un cuarto campo llamado LIGA AL PADRE. Por consiguiente la configuración del registro sería:

Liga Izquierda	Dato	Liga Derecha	Liga Al Padre

12.6 RECORRIDOS SOBRE ÁRBOLES BINARIOS

Una de las aplicaciones de árboles binarios es representar expresiones aritméticas. Si queremos representar la expresión **a+b** en un árbol binario tendríamos:

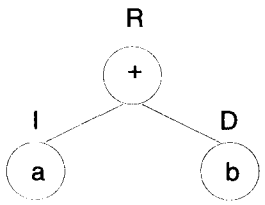


Figura 12.5

Es bueno observar que los operandos están en las hojas. Siempre que se desee representar una expresión en un árbol binario los operadores serán raíces y los operandos hojas.

Se deseamos recorrer el árbol de la figura 12.5 lo podemos hacer de varias maneras. Veamos:

	Recorrido	Resultado obtenido
1.	IRD	a + b
2.	IDR	a b +
3.	RID	+ a b
4.	RDI	+ b a
5.	DRI	b + a
6.	DIR	b a +

En las tres primeras formas obtenemos la expresión en **IN**fijo, **POS**fijo y **PRE**fijo respectivamente. Por analogía a la forma como se obtiene la expresión aritmética a estas primeras tres formas de recorrido se les han asignado los nombre de **INORDEN, POSORDEN y PREORDEN.**

Las formas 4, 5 y 6 no están definidas para recorrer árboles binarios. En general, digamos que las llamaremos, a forma de chiste, en DESORDEN.

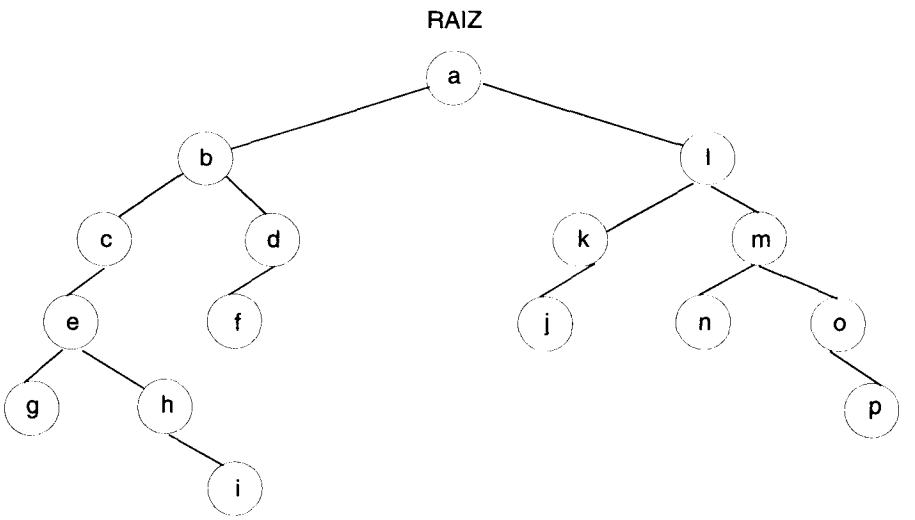
Nótese que los prefijos IN, POS y PRE hacen referencia a la ubicación de la raíz en cada recorrido.

Resumiendo tenemos:

1. **IRD (Izquierdo, Raíz, Derecho → INORDEN**
2. **IDR (Izquierdo, Derecho, Raíz) → POSORDEN**
3. **RID (Raíz, Izquierdo, Derecho) → PREORDEN**

Es bueno hacer notar que en todos los recorridos definidos siempre se visita el hijo izquierdo antes que el derecho. A continuación damos un ejemplo más amplio de los recorridos sobre una árbol binario.

Consideremos el siguiente árbol binario:



Recorrido INORDEN : I R D → gehicbfdajklnmop
Recorrido POSORDEN : I D R → gihecfdbjknpomla
Recorrido PREORDEN : R I D → abceghidflkjmnop

Algoritmos para ejecutar los recorridos

Para un árbol representado como lista ligada, el algoritmo recursivo para recorrerlo en INORDEN es:

```
sub_programa inorden(r)
      if r <> 0 then
            inorden(Li(r))
            write(dato(r))
            inorden(Ld(r))
      end(if)
fin(inorden)
```

El algoritmo sólo imprime el recorrido INORDEN sobre un árbol binario. La filosofía del algoritmo es: podré hacer recorrido cuando el árbol binario no sea

vacío (**if r <> 0**); antes de imprimir el dato de la raíz debo recorrer, también en INORDEN, la rama izquierda de la raíz (primera llamada recursiva); luego de imprimir el dato de la raíz debo recorrer, también en INORDEN, la rama derecha de la raíz (segunda llamada recursiva).

Si deseamos efectuar alguna operación diferente a imprimir el recorrido del árbol basta con reemplazar la instrucción WRITE por la instrucción o grupo de instrucciones correspondientes a lo que deseemos efectuar cuando estemos ubicados en ese registro.

A continuación presentamos los algoritmos correspondientes a los recorridos POSORDEN y PREORDEN.

```
sub_programa posorden(r)
      if r <> 0 then
            posorden(Li(r))
            posorden(Ld(r))
            write(dato(r))
      end(if)
fin(posorden)

sub_programa preorden(r)
      if r <> 0 then
            write(dato(r))
            preorden(Li(r))
            preorden(Ld(r))
      end(if)
fin(preorden)
```

12.7 REPRESENTACIÓN DE ÁRBOLES BINARIOS COMO LISTAS LIGADAS ENHEBRADAS

Cuando representamos un árbol binario como lista ligada, todos los registros que son hojas tendrán dos campos de liga cuyo valor es cero (nil), además aquellos registros que sólo tienen un hijo su otro campo de liga es cero (nil).

Consideremos el siguiente árbol binario en su representación como lista ligada:

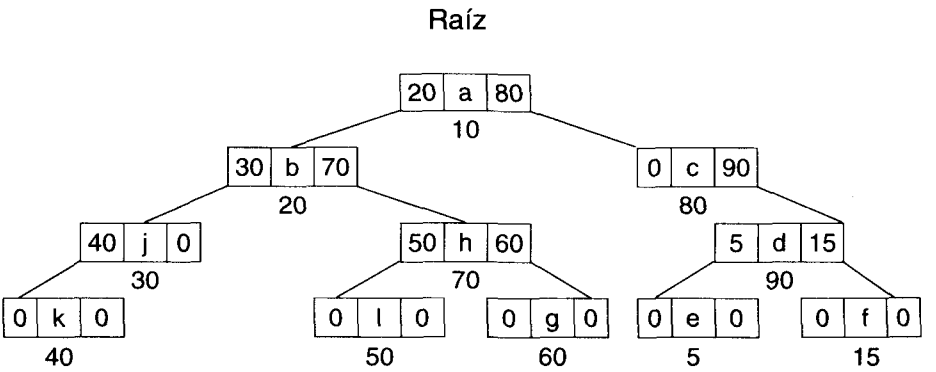


Figura 12.6

Dicho árbol tiene 11 registros, o sea 22 campos de liga, de los cuales hay 12 campos de liga que son cero. En general podemos afirmar que en un árbol binario, representado como lista ligada, que tenga **n** registros, habrá **n+1** campos de liga que son cero.

Como los campos cuyo valor es cero se consideran desperdicio de memoria se ha buscado la forma de utilizarlos de tal forma que redunden favorablemente ya sea en la elaboración de los algoritmos o en la eficiencia de ellos.

Consideremos el árbol de la figura 12.6 y escribamos su recorrido en INORDEN:

kjbihgacedf

el cual, si lo escribimos con los registros, en vez de con los datos que hay en cada registro tendremos:

40, 30, 20, 50, 70, 60, 10, 80, 5, 90, 15

es decir, ese es el orden en que pasamos por los registros.

La utilización que daremos a los campos de liga es: si hay un campo de liga izquierda que valga 0 lo pondremos a apuntar hacia el registro anterior en el recorrido INORDEN; si hay un campo de liga derecha que valga 0 lo pondremos a apuntar hacia el registro siguiente en recorrido INORDEN. El árbol quedará como en la figura 12.7

Note que el campo de liga izquierda del primer registro en recorrido INORDEN (el registro 40) sigue valiendo 0; igual cosa sucede con el campo de liga derecha del último registro en recorrido INORDEN (el registro 15).

Para obviar este problema lo que hacemos es añadir un registro cabeza al árbol. Entonces el campo de liga izquierda del primer registro apuntará hacia el registro cabeza y el campo de liga derecha del último registro también apuntará hacia el registro cabeza.

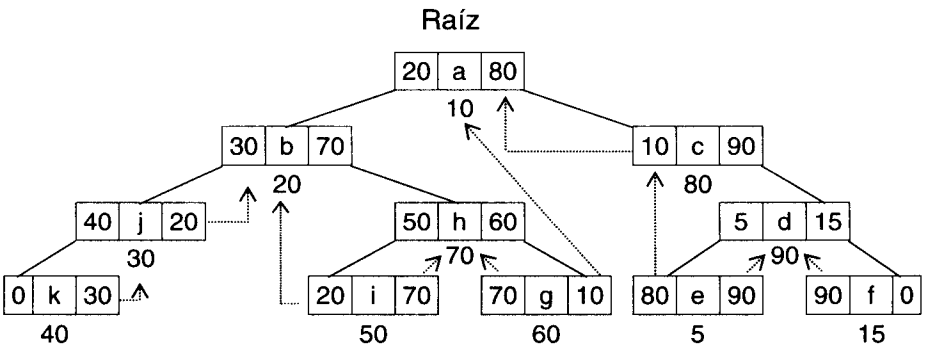


Figura 12.7

Nuestro árbol quedará como en la figura 12.8.

En el registro cabeza el campo de liga izquierda apuntará hacia la raíz del árbol, mientras que el campo de liga derecha apuntará hacia sí mismo.

Cuando tenemos el árbol representado de esta forma se nos presenta un nuevo problema. Si consideramos un campo de liga izquierda, por ejemplo, debemos reconocer cuando apunta hacia su hijo izquierdo o cuando apunta hacia el registro anterior en recorrido INORDEN. Es decir, el campo de liga izquierda tiene dos significados; con el campo de liga derecha sucede exactamente lo mismo.

A los campos de liga que apuntan hacia el registro anterior o hacia el siguiente se les denominan **hebras.**

Para distinguir cuando un campo de liga es una hebra o no, añadimos dos campos (que pueden ser bits) más al registro: Bit izquierdo y Bit derecho. El registro quedará entonces así:

Liga Izquierda	Bi	Dato	Bd	Liga Derecha

BI = 0: Li es una hebra (apunta hacia el registro anterior)

1: Li apunta hacia el hijo izquierdo.

BD = | 0: Ld es una hebra (apunta hacia el registro siguiente)
| 1: Ld apunta hacia el hijo derecho.

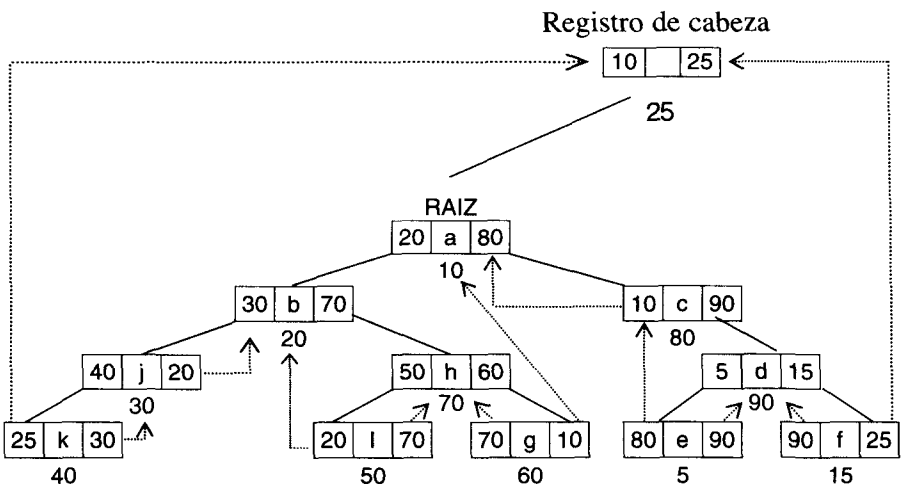
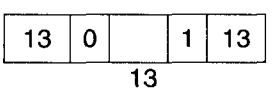


Figura 12.8

Para el registro cabeza: Bd siempre será 1
Ld siempre apunta hacia sí mismo.

y además,

Bi | 1 : Li apunta hacia la raíz del árbol
| 0 : árbol vacío y Li apunta hacia sí mismo.



Registro Cabeza de un árbol binario vacío

La representación de nuestro árbol con los registros completos, incluyendo los bits de diferenciación de ligas se presenta en la figura 12.9.

Con la representación de un árbol binario como lista ligada enhebrada se podrá saber fácilmente cual es el siguiente registro en recorrido INORDEN, para cualquier registro **x** perteneciente al árbol.

Si Bd(x) = 0 → Siguiente(x) = Liga_derecha(x)

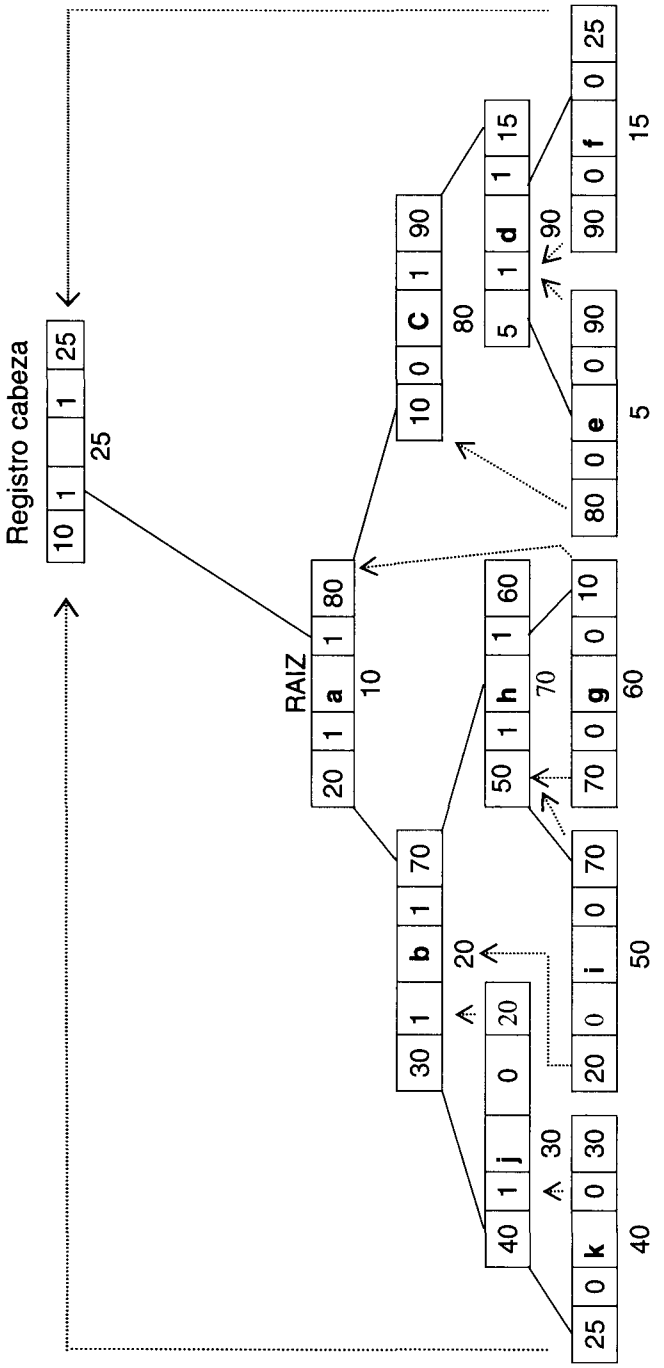

Registro cabeza
10 1 1 25
25
RAIZ
20 1 a 80
10
30 1 b 70
20
40 1 j 0 20
30
40 25 0 k 0 30
50 20 0 i 0 70
60 70 0 g 0 10
70 50 1 h 1 60
80 10 0 C 1 90
90 5 1 d 1 15
5 80 0 e 0 90
15 90 0 f 0 25

Figura 12.9

Si Bd(x) = 1 → Nos pasamos al hijo derecho de **x** y a partir de ese registro avanzamos con Liga_izquierda(x) hasta encontrar un registro con Bit izquierdo igual a cero. Ese registro es siguiente(x).

Elaboremos una función que ejecute dicha tarea:

```
function siguiente(x)
        siguiente = Ld(x)
        if Bd(x) = 1 then
                while Bi(siguiente) = 1 do
                        siguiente = Li(siguiente)
                end(while)
        end(if)
        return(siguiente)
fin(siguiente)
```

Teniendo dicha función podremos recorrer en INORDEN el árbol binario sin necesidad de utilizar recursión.

Dicho algoritmo es el siguiente:

```
sub_programa inorden (r)
        p = siguiente (r)
        while p <> r do
                write(dato (p))
                p=siguiente(p)
        end(while)
fin(inorden)
```

12.8 ÁRBOLES BINARIOS DE BÚSQUEDA

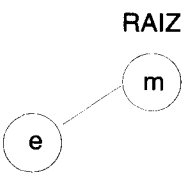
Son árboles binarios en los cuales se cumple que para cualquier registro **x** perteneciente al árbol, todos los datos de los registros a la izquierda de **x** son menores que el dato de **x** y todos los datos de los registros a la derecha de **x** son mayores que el dato de **x**.

Consideremos el siguiente conjunto de datos: m e v c x l p a i s

Construyamos un árbol binario de búsqueda con ellos.

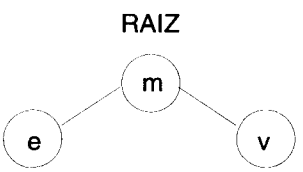
Comencemos leyendo el primer dato, es decir, **m**. Como el árbol está vacío simplemente conseguimos un registro, le asignamos el dato **m** y decimos que este registro es la raíz:

Luego leemos el siguiente dato, es decir **e**. Como ya empezamos a construir el árbol hay que buscar dónde insertarlo. Para ello, comparamos el dato de la raíz con el dato leído. Si el dato leído es mayor que el dato de la raíz avanzamos por la rama derecha de la raíz, de lo contrario avanzamos por la rama izquierda de la raíz. En nuestro caso avanzamos por el lado izquierdo de la raíz, como no tiene hijo izquierdo simplemente conseguimos registro y lo insertamos como hijo izquierdo de la raíz. Nuestro árbol queda:



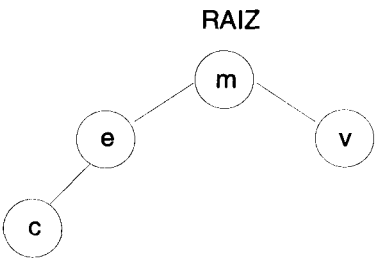
Continuamos leyendo el siguiente dato, es decir, **v.**

Debemos buscar dónde insertarlo. Empezamos comparando el dato leído con el dato de la raíz. Si el dato leído es mayor que el dato de la raíz avanzamos por la rama derecha de la raíz, en caso contrario por el lado izquierdo. En nuestro caso avanzaremos por el lado derecho. Como no existe hijo derecho simplemente conseguimos un registro y lo insertamos como hijo derecho de la raíz. Nuestro árbol quedará:

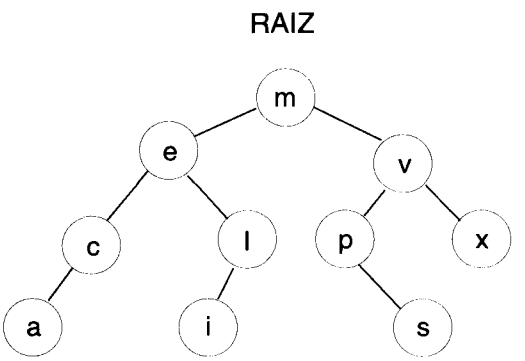


Leemos el siguiente dato que la **c.** Habrá que buscar dónde insertarlo.

Comenzamos comparando el dato leído con el dato de la raíz. Como el dato leído es menor que el dato de la raíz avanzamos por la rama izquierda de la raíz. Quedamos en el registro que tiene la **e**, comparamos el dato leído con este dato, como el dato leído es menor avanzamos por la rama izquierda, como dicho registro no tiene hijo izquierdo simplemente conseguimos un registro y lo insertamos como hijo izquierdo del registro que tiene la **e**. Nuestro árbol quedará:



Continuando esta metodología, de añadir un registro al árbol binario, por cada dato que se lea, nuestro árbol quedará así:



Todos los datos de la izquierda de cualquier registro son menores que el dato de ese registro, y todos los de la derecha son mayores. Además si hacemos un recorrido INORDEN sobre dicho árbol obtenemos los datos ordenados ascendentemente:

aceilmpvsx

12.9 ÁRBOLES AVL

Son árboles binarios balanceados por altura. Su nombre se debe a que sus inventores son los soviéticos **A**delsson, **V**enski y **L**andis.

Factor de balance

Es el concepto clave para definir los árboles AVL. El factor de balance para cualquier registro **x** se define como la diferencia entre la altura del hijo izquierdo de **x** y la altura del hijo derecho de **x**. Es decir:

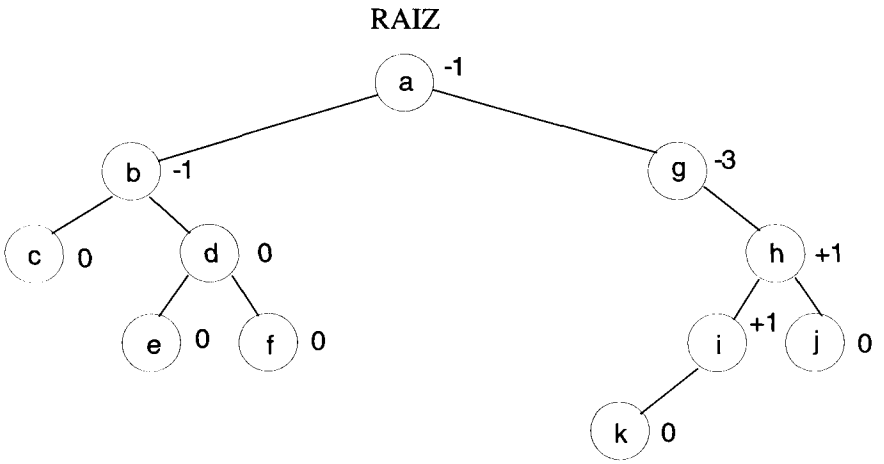
Fb(x) = Altura(Li(x)) - Altura(Ld(x)).

Un árbol binario es AVL si: $|\mathbf{FB(X)}| < \mathbf{2}$

o sea el conjunto de valores permitidos en el factor de balance, para que un árbol binario sea AVL es:

$$\mathbf{FB = \{-1, 0, +1\}}$$

Consideremos el siguiente árbol, al cual le hemos colocado, al lado de cada registro, el factor de balance correspondiente.



Dicho árbol no es AVL ya que el registro que contiene el dato **g** tiene factor de balance -3.

Nuestro objetivo será construir árboles AVL.

Rebalancear un árbol

Consiste en reacomodar los registros de un árbol binario de tal forma que los factores de balance de todos los registros sean **-1, 0, ó +1** y que el recorrido INORDEN sea el mismo que antes del reacomodo.

Operaciones de rebalanceo

1. Una rotación a la derecha.
2. Una rotación a la izquierda.
3. Doble rotación a la derecha.
4. Doble rotación a la izquierda.

Para explicar lo referente a las rotaciones asumamos la siguiente convención:

Sea **P** la dirección del registro con factor de balance no permitido.(+2 ó -2).

Sea **Q** la dirección del hijo izquierdo o del hijo derecho de **P,** dependiendo de si Fb(P)=+2 ó Fb(P)= - 2, es decir, si factor de balance de **P** es +2 entonces **Q** es el hijo izquierdo de **P** y si factor de balance de **P** es -2 entonces **Q** es el hijo derecho de **P**.

1. Una rotación a la derecha

Se efectúa cuando: Fb (P) = + 2 y Fb (Q) = + 1

Consiste en girar, en sentido de las manecillas del reloj, el registro **P** alrededor del registro **Q**.

Consecuencias:

- **P** pasará a ser el nuevo hijo derecho de **Q**.
- El anterior hijo derecho de **Q** será el nuevo hijo izquierdo de **P**.
- **Q** será la nueva raíz del árbol balanceado.
- Los nuevos factores de balance de **P** y **Q** serán cero (0).
- La altura del árbol balanceado disminuye en uno (1).

En las figuras 12.10a y 12.10b mostramos un árbol que se acomoda a esta primera situación: antes y después del balanceo.

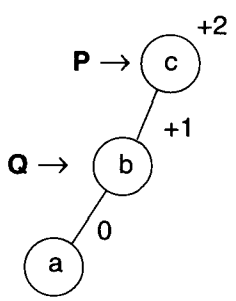


Figura 12.10a

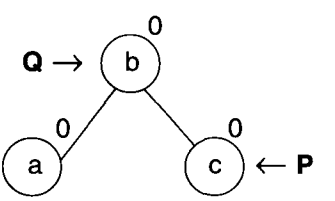


Figura 12.10b

Note además que los recorridos INORDEN sobre ambos árboles es el mismo, antes y después de la rotación.

El algoritmo que efectúa las operaciones descritas anteriormente es el siguiente:

```
sub_programa una_rot_a_la_derecha(p, q)
      Li(p) = Ld(q)
      Ld(q) = p
      Fb(p) = 0
      Fb(q) = 0
fin(una_rot_a_la_derecha)
```

En las figuras 12.11a y 12.11b presentamos otro ejemplo del rebalanceo en el caso de una rotación a la derecha. La figura 12.13a es antes del rebalanceo y la figura 12.13b es después del rebalanceo.

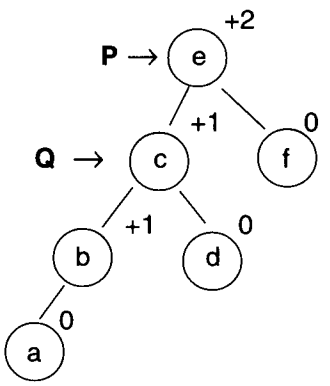


Figura 12.11a

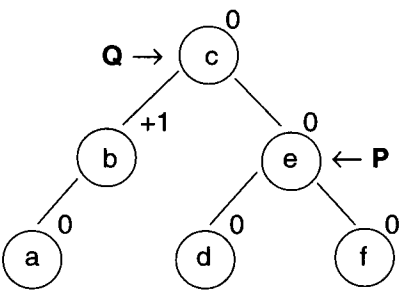


Figura 12.11b

Note que los recorridos INORDEN son iguales en ambos árboles.

Fíjese que el registro **P** pasó a ser el hijo derecho del registro **Q**; el registro que tiene el dato **d**, que era el hijo derecho del registro **Q**, pasó a ser el hijo izquierdo del registro **P**; el registro **Q** es la nueva raíz del árbol balanceado, y los factores de balance de los registros **P** y **Q** quedaron en cero.

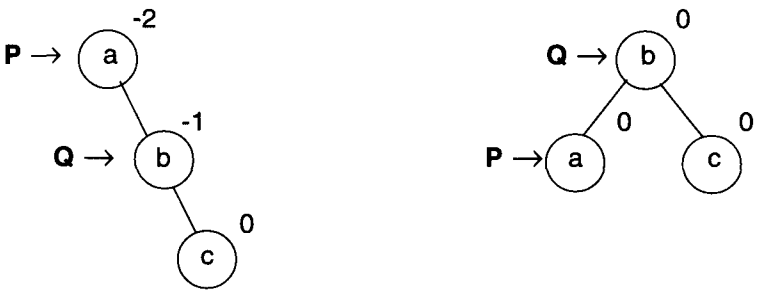
2. Una Rotación a la Izquierda

Se efectúa cuando	Fb(P) = - 2 y Fb(Q) = - 1

Consiste en girar, en sentido contrario de las manecillas del reloj, **P** alrededor de **Q**.

Consecuencias:

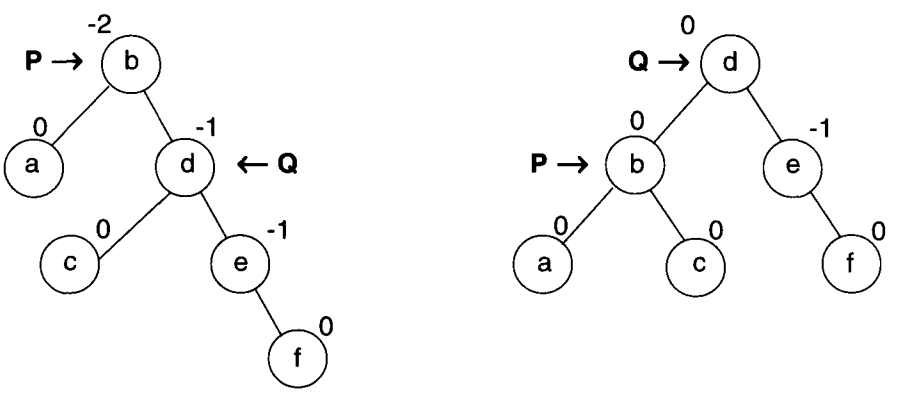
- **P** será el nuevo hijo izquierdo de **Q**.
- El nuevo hijo derecho de **P** será el anterior hijo izquierdo **Q**.
- **Q** será la nueva raíz del árbol balanceado.
- Los factores de balance **P** y **Q** quedarán en cero.
- La altura del árbol balanceado disminuye en uno (1).



El algoritmo que ejecuta las operaciones descritas es el siguiente:

```
sub_programa una_rot_a_la_izquierda(p, q)
        Ld(p) = Li(q)
        Li(q) = p
        Fb(p) = 0
        Fb(q) = 0
fin(una_rot_a_la_izquierda)
```

Consideremos otro ejemplo:



Note que los recorridos INORDEN son iguales en ambos árboles.

Fíjese que el registro **P** pasó a ser el hijo izquierdo del registro **Q**; el registro que tiene el dato **C**, que era el hijo izquierdo del registro **Q**, pasó a ser el hijo derecho del registro **P**; el registro **Q** es la nueva raíz del árbol balanceado, y los factores de balance de los registros **P** y **Q** quedaron en cero.

Para definir las dobles rotaciones adoptemos una nueva convención:

Sea **R** el registro que representa el hijo izquierdo o el hijo derecho de **Q** dependiendo de si el factor de balance de **Q** es +1 ó -1. Es decir, si el factor de balance del registro **Q** es +1, el registro **R** es el hijo izquierdo del registro **Q**, y si el factor de balance de **Q** es -1, **R** es el hijo derecho del registro **Q**.

3. Doble Rotación a la Derecha

Se efectúa cuando	Fb(P) = +2
	Fb(Q) = -1

Consiste en una rotación a la izquierda de **Q** alrededor de **R** seguida de una rotación a la derecha de **P** alrededor de **R**.

Consecuencias:

- **R** será la nueva raíz del árbol balanceado.
- **P** será el nuevo hijo derecho de **R**.
- **Q** será el nuevo hijo izquierdo de **R**.
- El anterior hijo derecho de **R** será el nuevo hijo izquierdo de **P**.
- El anterior hijo izquierdo de **R** será el nuevo hijo derecho de **Q**.
- La altura del árbol balanceado disminuye en uno (1).
- **R** será la nueva raíz del árbol balanceado.
- El factor de balance de **R** será cero.
- Los factores de balance de **P** y **Q** tomarán nuevos valores, los cuales dependerán del factor de balance inicial del registro **R**. El factor de balance inicial del registro **R** puede ser cero (0), más uno (+1) o menos uno (-1).

Consideremos el primer caso, cuando el factor de balance inicial del registro **R** es cero.

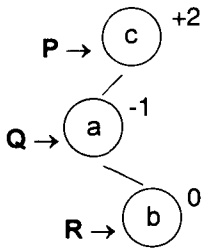


Figura 12.12a

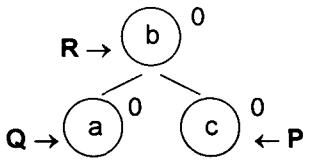


Figura 12.12b

En nuestro primer caso, figuras 12.12a y 12.12b, el factor de balance inicial del registro **R** es cero. Para este caso los factores de balance finales de los registros **P** y **Q** serán cero.

Si Fb inicial de **R** es cero | Fb(P) = 0 ; Fb(Q) = 0

Consideremos el segundo caso, cuando el factor de balance inicial del registro **R** es más uno (+1)

Las figuras 12.13a y 12.13b ilustran esta situación. Como podrá observarse, el factor de balance final del registro Q es cero (0), mientras que el factor de balance final del registro **P** es menos uno (-1).

Si Fb inicial de **R** es más uno (+1) | Fb(P) = -1 ; Fb(Q) = 0

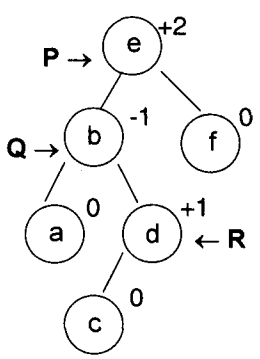


Figura 12.13b

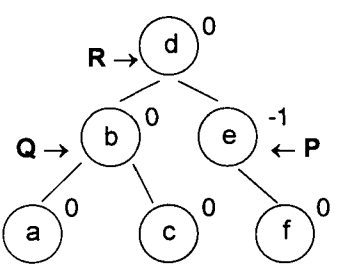


Figura 12.13a

Consideremos finalmente el caso en el cual el factor de balance inicial del registro **R** es menos uno (-1):

Las figuras 12.14a y 12.14b ilustran esta situación. Como podrá observarse, el factor de balance final del registro **P** es cero (0), mientras que el factor de balance final del registro **Q** es más uno (+1).

Si Fb inicial de **R** es menos uno (-1)	Fb(P) = 0 Fb(Q) = +1

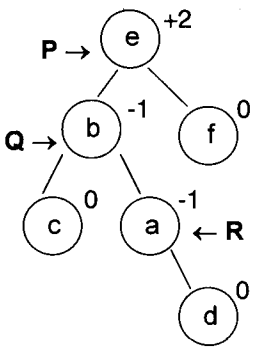


Figura 12.14a

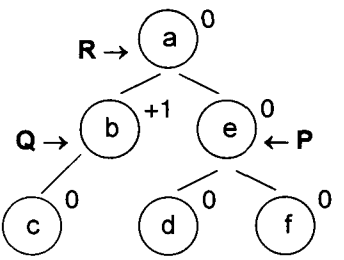


Figura 12.14b

En la siguiente página presentamos el algoritmo completo que ejecuta la doble rotación a la derecha. Es importante hacer notar que el parámetro **Q** debe ser un parámetro por referencia, ya que en él retornará la raíz del nuevo árbol balanceado. Esto con el fin de unificar los algoritmos de balanceo: en la variable **Q** tendremos la raíz del árbol balanceado.

```
sub_programa doble_rot_a_la_derecha(p, q)
        r = Ld(q)
        Ld(q) = Li(r)
        Li(r) = q
        Li(p) = Ld(r)
        Ld(r) = p
        Casos de Fb(r)
              :0:   Fb(p) = 0
                    Fb(q) = 0

              :1:     Fb(p) = -1
                    Fb(q) = 0
```

```
            :-1:  Fb(p) = 0
                  fb(q) = 1
        fin(casos)
        Fb(r) = 0
        q = r
fin(doble_rot_a_la_derecha)
```

4. Doble rotación a la izquierda

Se efectúa cuando | FB(P) = -2 ; FB(Q) = +1

Consiste en una rotación a la derecha de **Q** alrededor de **R** seguida de una rotación a la izquierda de **P** alrededor de **R**.

Consecuencias:

- **R** será la nueva raíz del árbol balanceado.
- **P** será el nuevo hijo izquierdo de **R**.
- **Q** será el nuevo hijo derecho de **R**.
- El anterior hijo derecho de **R** será el nuevo hijo izquierdo de **Q**.
- El anterior hijo izquierdo de **R** será el nuevo hijo derecho de **P**.
- La altura del árbol balanceado disminuye en uno (1).
- **R** será la nueva raíz del árbol balanceado.
- El factor de balance de **R** será cero, y los factores de balance de **P** y **Q** tomarán nuevos valores, los cuales dependerán del factor de balance inicial del registro **R**. El factor de balance inicial del registro **R** puede ser cero (0), más uno (+1) o menos uno (-1).

Consideremos el primer caso, cuando el factor de balance inicial del registro **R** es cero.

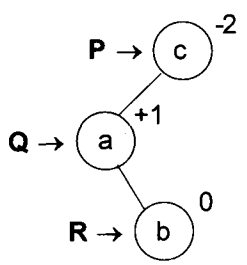


Figura 12.15a

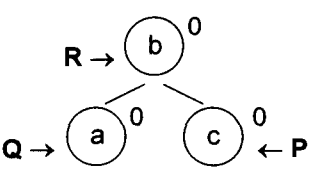


Figura 12.15b

En nuestro primer caso, figuras 12.15a y 12.15b, el factor de balance inicial del registro **R** es cero. Para este caso los factores de balance finales de los registros **P** y **Q** serán cero.

Si Fb inicial de **R** es cero | Fb(P) = 0 ; Fb(Q) = 0

Consideremos el segundo caso, cuando el factor de balance inicial del registro **R** es más uno (+1)

Las figuras 12.16a y 12.16b ilustran esta situación. Como podrá observarse, el factor de balance final del registro **P** es cero (0), mientras que el factor de balance final del registro **Q** es menos uno (-1).

Si Fb inicial de **R** es más uno (+1) | Fb(P) = 0 ; Fb(Q) = -1

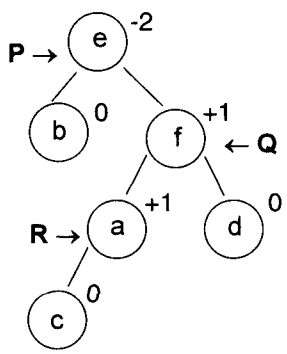


Figura 12.16a

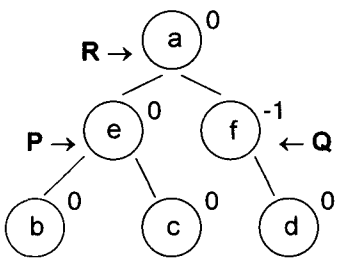


Figura 12.16b

Consideremos finalmente el caso en el cual el factor de balance inicial del registro **R** es menos uno (-1):

Las figuras 12.17a y 12.17b ilustran esta situación. Como podrá observarse, el factor de balance final del registro **Q** es cero (0), mientras que el factor de balance final del registro **P** es más uno (+1).

Si Fb inicial de **R** es menos uno (-1) | Fb(Q) = 0 ; Fb(P) = +1

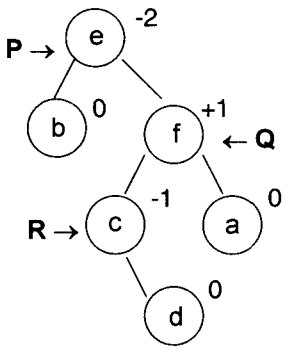


Figura 12.17a

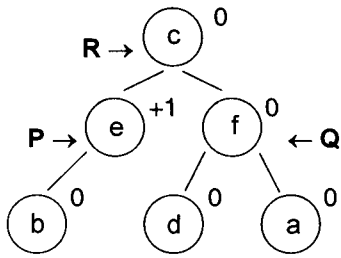


Figura 12.17b

A continuación presentamos el algoritmo completo que ejecuta la doble rotación a la izquierda. Es importante hacer notar que el parámetro Q debe ser un parámetro por referencia, ya que en él retornará la raíz del nuevo árbol balanceado.

```
sub_programa doble_rot_a_la_izquierda(p, q)
        r = Li(q)
        Li(q) = Ld(r)
        Ld(r) = q
        Ld(p) = Li(r)
        Li(r) = p
        Casos de  Fb(r)
                :0: Fb(p) = 0
                    Fb(q) = 0

                :1: Fb(p) = 0
                    Fb(q) = -1

                :-1: Fb(p) = 1
                    Fb(q) = 0
        fin(casos)
        Fb(r) = 0
        q = r
fin(doble_rot_a_la_izquierda)
```

Construcción de árbol AVL. Veamos ahora cómo construir un árbol binario de tal forma que cumpla las propiedades AVL.

Hay que tener en cuenta que en la construcción de un árbol AVL todo registro que se inserte, se insertará como una hoja. Los pasos a seguir son:

1. Buscar donde insertar el nuevo registro.
2. Insertar el nuevo registro.
3. Recalcular factores de balance.
4. Rebalancear el árbol si es necesario.

Consideremos el árbol de la figura 12.18 y que vamos a insertar un registro con el dato 77.

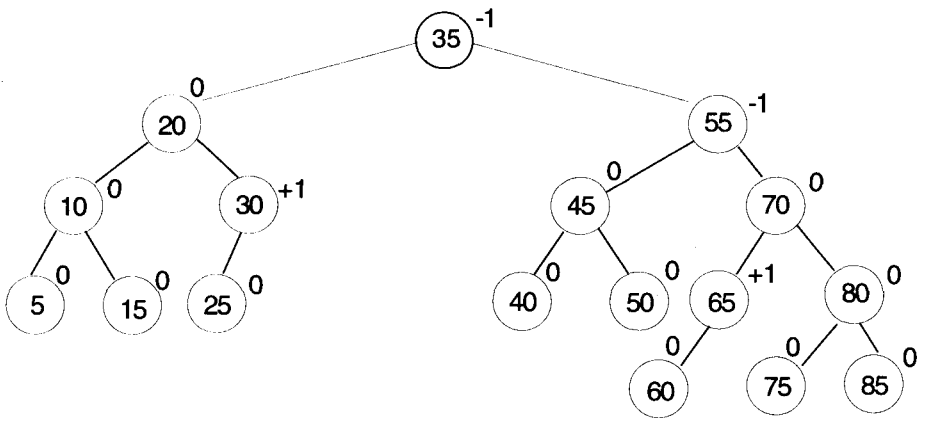


Figura 12.18

Los datos a la izquierda de cada registro son menores que el dato de ese registro y los de la derecha mayores.

Para el primer paso, el cual consiste en buscar dónde insertar el nuevo registro utilizaremos las siguientes variables:

P: Para recorrer el árbol buscando donde insertar el nuevo registro.
Q: Apuntará hacia el padre de **P**.

El primer paso consiste en comparar el dato de un registro con el dato leído. En caso de que el dato de ese registro sea mayor que el dato leído avanzaremos por la rama izquierda del registro con el cual se hizo la comparación, de lo contrario avanzaremos por la rama derecha. Cuando **P = 0**, Q apuntará hacia el registro que será el padre del registro que contiene el nuevo dato.

Inicialmente P será la raíz y Q será cero (0).

Ahora bien, como el tercer paso es rebalancear el árbol, se requiere conocer cuál será el registro "raíz" del sub-árbol a balancear. Llamaremos este registro PIVOTE. Es decir, PIVOTE será la raíz del árbol a balancear.

Como al rebalancear un árbol, éste siempre cambia de raíz, la nueva raíz, que es **Q**, habrá que pegarla del registro que era padre de PIVOTE. Esto implica que debemos conocer el padre de PIVOTE. Llamaremos este registro **PP**.

Resumiendo tenemos:

PIVOTE: Dirección del registro que posiblemente quede desbalanceado como consecuencia de la inserción.
PP: Dirección del registro padre de PIVOTE

Para determinar PIVOTE debemos tener en cuenta que los únicos registros que pueden quedar desbalanceados son aquellos cuyo factor de balance es diferente de cero (0). En algunos casos puede suceder que PIVOTE sea RAIZ, con factor de balance cero. En concreto, el registro que será PIVOTE es el registro más cercano al registro donde se hará la inserción, con factor de balance diferente de cero. Es bueno aclarar que es el registro más cercano en la trayectoria desde la raíz hasta el registro que será el nuevo padre.

Inicialmente PIVOTE será la raíz y PP será cero (0).

El algoritmo de construcción quedará así:

```
sub_programa consavl(r, d)
    new(x)
    dato(x) = d          Consigue nuevo registro y lo configura
    Li(x) = 0            con el dato a insertar
    Ld(x) = 0
    Fb(x) = 0
    if r = 0 then        | Si es la primer vez que se llama el
        r = x            | procedimiento, el registro conseguido
        return           | es la raíz.
    end(if)
    p = r                Asigna valores iniciales a las variables
    q = 0                que se utilizan para buscar el sitio donde
    pivote = r           se insertará nuevo registro y determinar
    pp= 0                PIVOTE y su padre.
    while p <> 0 do
        if dato(p) = d then
            free(x)                  | Controla que no vayan a
            return                   | quedar datos repetidos
        end(if)
        If Fb(p) <> 0 then
```

```
            pivote = p          | Determina valor de PIVOTE
            pp = q              | y de su padre.
      end(if)
      q = p
      if dato(p) > d then       | Actualiza Q
            p = Li(p)           | y
      else                      | avanza con P
            p = Ld(p)
      end(if)
end(while)
if dato(q) > d then
      Li(q) = x                 | Inserta x como hijo izquierdo
else                            | o como hijo derecho de Q
      Ld(q) = x
end(if)
if dato(pivote) > d then
    Fb(pivote) = Fb(pivote)+1
     q = Li(pivote)                    | Recalcula factor de balance
else                                   | de PIVOTE
      Fb(pivote = Fb(pivote) – 1
      q = Ld(pivote)
end(if)
p = q
while p <> x do
      if dato(p) > d then
            Fb(p) = +1          | Recalcula factores de balance
            p = Li(p)           | de todos los registros en la
      else                      | trayectoria desde PIVOTE hasta
            Fb(p) = -1          | el registro insertado X
            p = Ld(p)
      end(if)
end(while)
if abs(Fb(pivote)) < 2 then           | Si el árbol siguió balanceado
      return                          | retorna al programa llamante.
end(if)
if Fb(pivote) = +2 then
      if Fb(q) = +1 then
            una_rot_a_la_derecha(pivote, q)       | Determina qué
      else                                        | tipo de
            doble_rot_a_la_derecha(pivote, q)     | rotación hay
      end(if)                                     | que efectuar
else                                              | y llama
      if Fb(q) = -1 then                          | el
            una_rot_a_la_izquierda(pivote, q)     | procedimiento
      else                                        | correspondiente
```

```
                doble_rot_a_la_izquierda(pivote, q)
            end(if)
        end(if)
        if pp = 0 then          | Si el registro desbalanceado
            r = q               | era la raíz actualiza la nueva
            return              | raíz y regresa.
        end(if)

        if pivote = li(pp) then
            Li(pp) = q          | Pega la nueva raíz del árbol
        else                    | rebalanceado al registro Q.
            Ld(pp) = q
        end(if)
fin(consavl)
```

12.10 REPRESENTACIÓN DE UN ÁRBOL NO BINARIO COMO BINARIO

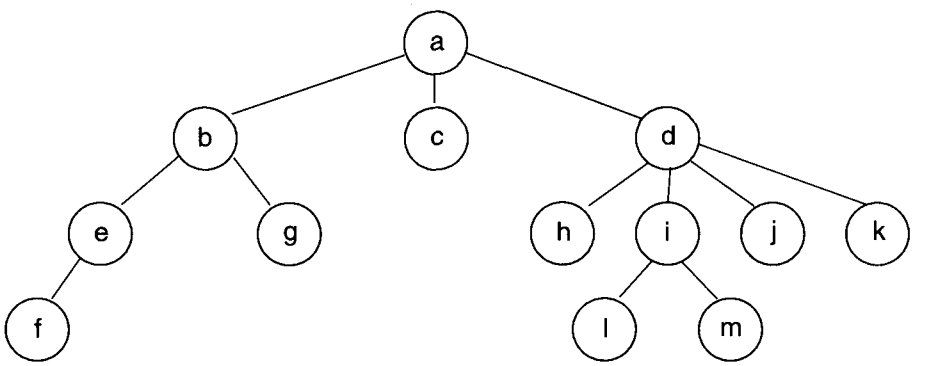
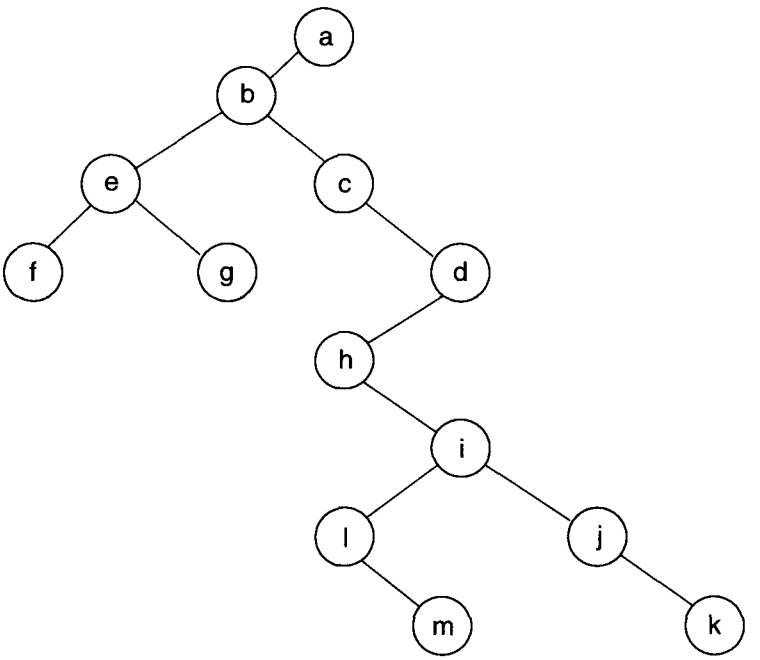


Figura 12.19

Cualquier árbol, sin importar su grado puede representarse en árbol binario.

Veamos la representación binaria equivalente al árbol de la figura 12.19.

La construcción del árbol binario se hace de la siguiente manera: para un registro que tenga **n** hijos, su primer hijo se representa como hijo izquierdo y los demás registros hermanos se insertan como hijos derechos del primero. Es decir, los registros que estén en una rama derecha son hermanos del primer registro de esa rama.



12.11 CONSTRUCCIÓN DE UN ÁRBOL BINARIO DADOS LOS RECORRIDOS PREORDEN E INORDEN O POSORDEN E INORDEN

Conocidos los recorridos PREORDEN e INORDEN o POSORDEN e INORDEN sobre un árbol binario podremos construir el árbol que cumple dichos recorridos. Por ejemplo, para el árbol de la figura 12.21 los recorridos PREORDEN e INORDEN son:

PREORDEN a b d h i m e c f j n k g l
INORDEN h d m i b e a j n f k c g l

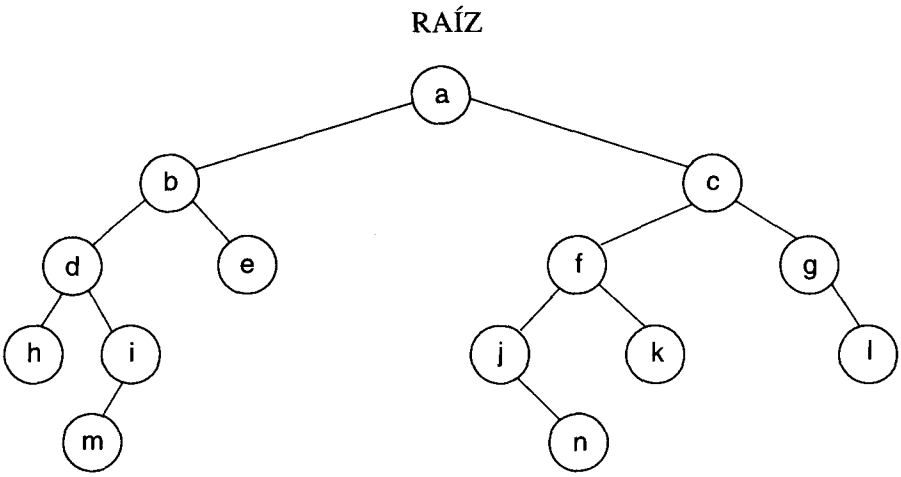


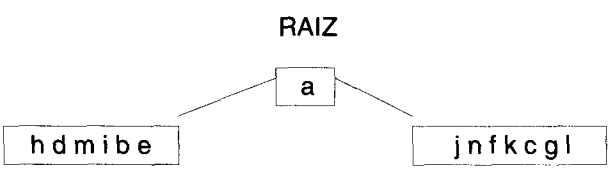
Figura 12.20

Pero si desconocemos el árbol, podemos construirlo siempre y cuando conozcamos sus recorridos Preorden e Inorden (ó Posorden e Inorden) como se ilustra a continuación.

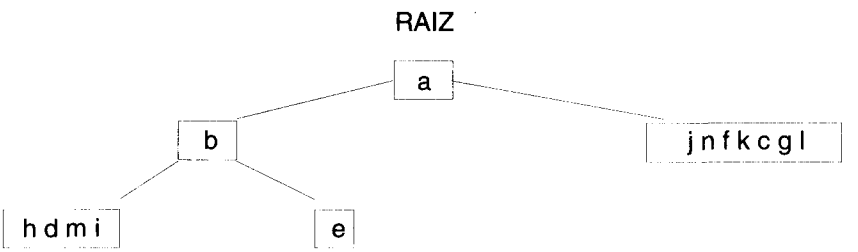
Si consideramos nuevamente los recorridos PREORDEN e INORDEN tenemos:

PREORDEN **a** b d h i m e c f j n k g l
INORDEN h d m i b e **a** j n f k c g l

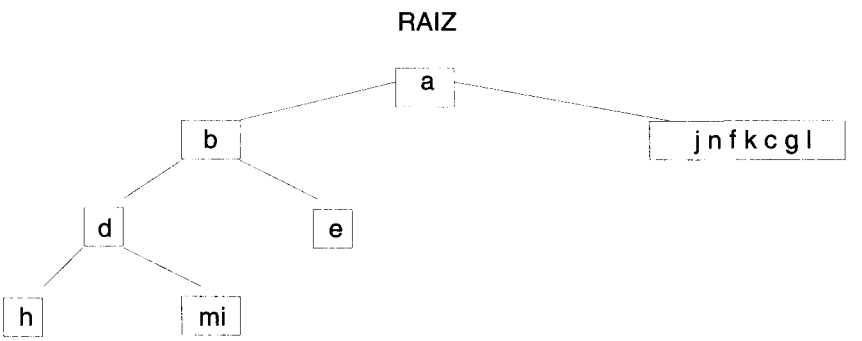
En el recorrido PREORDEN el primer registro que se visita es el registro que contiene el dato **a**. Bastará entonces con ubicar la posición de este dato en el recorrido INORDEN, y todos los datos que queden antes de la **a** corresponden al recorrido INORDEN del sub-árbol binario que es hijo izquierdo de **a**, los datos que queden a la derecha de **a** corresponden al recorrido INORDEN del sub-árbol que es hijo derecho de **a**. Por consiguiente un primer bosquejo del árbol a construir es el siguiente:



Continuando con el recorrido PREORDEN encontramos el dato **b**. El paso a seguir es ubicarlo en el sub-árbol izquierdo de **a**; los datos que queden a la izquierda de **b** son el recorrido INORDEN del subárbol que es hijo izquierdo de **b**, los dato a la derecha son el recorrido INORDEN del sub-árbol que es hijo derecho de **b**. Nuestro árbol quedará así:



Continuando con el recorrido PREORDEN inicial encontramos el dato **d**, ubicamos **d** en el sub-árbol que es hijo izquierdo de **b** y procedemos de una manera análoga a los casos anteriores. Nuestro árbol quedará así:



Continuando con este proceso, el cual a todas luces es recursivo, obtenemos el árbol inicial mostrado en la figura 12.21.

Un algoritmo que efectúe esta tarea es:

```
function consab(lin, lis, i)
1.    hi = 0
2.    hd = 0
3.    k = lin
4.    if  lis > lin  then
```

```
5.          while  pre(i) <> inor(k)  do
6.                 k = k + 1
7.          end(while)
8.          if  k > lin  then
9.                 hi = cab(lin, k – 1, i + 1)
10.                if  k < lis  then
11.                       hd = cab(k+1, lis, k+1)
12.                end(if)
13.         else
14.                if  k < lis  then
15.                       hd = cab(k+1, lis, i+1)
16.                end(if)
17.         end(if)
18.   end(if)
19.   new(x)
20.   dato(x) = inor(k)
21.   Li(x) = hi
22.   Ld(x) = hd
23.   return(x)
fin(consab)
```

Los parámetros **lin** y **lis** son límite inferior y límite superior respectivamente. Indican el rango sobre el cual hay que buscar el dato **pre(i)** en el vector **inor**. El parámetro **i** se utiliza para recorrer el vector **pre**.

Los vectores **pre** e **inor** son variables globales en las cuales se almacenan los recorridos preorden e inorden del árbol que se va a construir.

Dejamos al lector como ejercicio comprobar la correctitud de dicho algoritmo.

EJERCICIOS PROPUESTOS

1. Escriba un algoritmo para determinar el número de hojas de un árbol binario representado como lista ligada.

2. Escriba un algoritmo para determinar el grado de un árbol binario representado como lista ligada.

3. Escriba un algoritmo para determinar la altura de un árbol binario representado como lista ligada.

4. Escriba un algoritmo para determinar el número de hojas de un árbol representado como lista generalizada.

5. Escriba un algoritmo para determinar el grado de un árbol representado como lista generalizada.

6. Escriba un algoritmo para determinar la altura de un árbol representado como lista generalizada.

8. Elabore un algoritmo para construir un árbol binario al cual se le dieron los recorridos POSORDEN e INORDEN.

9. Elabore algoritmos para determinar el número de hojas, el grado y la altura de un árbol teniéndolos representados como listas ligadas enhebradas.

10. Elabore algoritmos para recorrer árboles binarios representados como listas ligadas enhebradas.

11. Elabore un algoritmo que borre un registro de un árbol AVL.

12. Escriba un algoritmo para determinar el padre de un registro en un árbol binario representado como lista ligada enhebrado.

13. Escriba un algoritmo para determinar el padre de un registro en un árbol binario representado como lista ligada.

14. Escriba un algoritmo para determinar el padre de un registro en un árbol binario representado como lista generalizada.

15. Escriba un algoritmo que construya la representación de una árbol como lista generalizada, dada la entrada como una hilera de paréntesis izquierdos, átomos, comas y paréntesis derechos.

16. Escriba un algoritmo que construya la representación como árbol binario en lista ligada de un árbol cualquiera, dada su representación como lista generalizada.

13

GRAFOS

13.1 DEFINICIÓN

Un grafo consiste en:

- Un conjunto finito de vértices.
- Un conjunto de pares de vértices llamados lados.

Ejemplos:

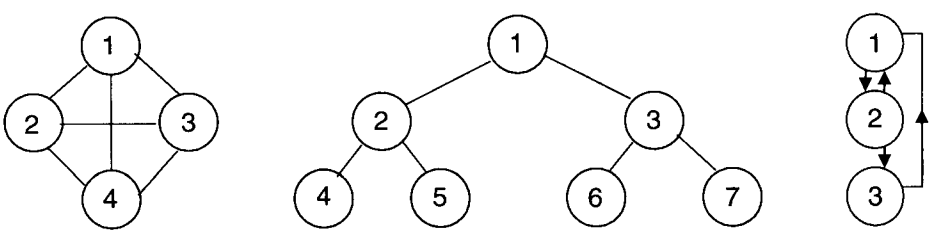


Figura 13.1

Los conjuntos de vértices y lados para cada uno de los grafos de la figura 13.1 son:

Grafo 1 (**G1**) :
V = {1,2,3,4}
E = {(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)}

Grafo 2 (**G2**) :
V = {1,2,3,4,5,6,7}
E = {(1,2), (1,3), (2,4), (2,5), (3,6), (3,7)}

Grafo 3 (**G3**) :
V = {1,2,3}
E = {<1,2>, <2,1>, <2,3>, <3,1>}

13.2 CLASIFICACIÓN

Los grafos se clasifican en:

a) **Grafos no Dirigidos:** son aquellos en los cuales los lados no están orientados, como en los grafos **G1** y **G2**. Cuando se trata de grafos no dirigidos cada pareja de vértices se representa entre paréntesis. El orden de los vértices que definen un lado no es relevante, es decir el lado (V_i, V_j) es exactamente lo mismo que el lado (V_j, V_i).
b) **Grafos Dirigidos:** son aquellos grafos en los cuales los lados están orientados como en el grafo **G3**. La pareja de vértices que representa un lado se escribe entre ángulos. La orientación del lado depende del orden en que escriba la pareja de vértices. Es decir, el lado $<V_i, V_j>$ es diferente del lado $<V_i, V_j>$.

Cuando se trata de un grafo dirigido, cada vértice de un lado se diferencia así:

$$<V_i, V_j> \begin{cases} V_i = \text{cola del lado} \\ V_j = \text{cabeza del lado} \end{cases}$$

13.3 TERMINOLOGÍA

Definamos ahora la terminología que utilizaremos en grafos.

Dos vértices son **adyacentes** si conforman un lado.

En el grafo **G1** los vértices 1 y 2 son adyacentes, también lo son los vértices 2 y 3.

Para grafos dirigidos la relación de adyacencia se discrimina así: Si tenemos el lado $<V_i, V_j>$ se dice que V_i **es adyacente hacia** V_j y que V_j **es adyacente desde** V_i.

El lado formado por dos vértices es **incidente** sobre ellos.

Grado de un Vértice. Es el número de lados incidentes sobre él. Para el grafo **G1** el grado del vértice 1 es 3. Para el grafo **G2** el vértice 5 tiene grado 1 y el vértice 3 tiene grado 3. Para el grafo **G3** el vértice 2 tiene grado 3.

Para grafos Dirigidos hay que diferenciar entre **Grado Entrante** y **grado saliente.**

Grado entrante es el número de lados que llegan al vértice y **Grado Saliente** es el número de lados que salen del vértice.

Para el grafo **G3** el vértice 1 tiene: Grado entrante = 2 y Grado saliente = 1

En grafos dirigidos el grado total es la suma del grado entrante más el grado saliente.

Trayectoria. Son los vértices por los cuales hay que pasar para ir desde un vértice **i** hacia un vértice **j**. Por ejemplo en el grafo G2 para ir desde el vértice 4 hacia el 6, la trayectoria es:

4, 2, 1, 3, 6 {1}

Para que una trayectoria sea válida la condición es que los lados sobre la trayectoria existan en el conjunto de lados que define el grafo. En nuestro ejemplo, los lados definidos en la trayectoria son los lados (4,2), (2,1), (1,3) y (3,6). Todos esos lados existen en el conjunto de lados que definen el grafo **G2**, por tanto dicha trayectoria es válida.

Si tuviéramos otra trayectoria: 4, 2, 3, 6

Los lados definidos en ella son: (4,2), (2,3) y (3,6). Si observamos el conjunto de lados que define el grafo **G2**, allí no se encuentra definido el lado (2,3), por consiguiente, la trayectoria presentada aquí no es válida.

Longitud de una Trayectoria: es el número de lados en ella. En el ejemplo de la trayectoria {1}, la cual, es válida, en el grafo **G2,** para ir desde el vértice cuatro hacia el vértice seie es 4, es decir hay que pasar por cuatro lados.

Trayectoria Simple: es una trayectoria en la que todos los vértices, excepto posiblemente el primero y el último, son diferentes. Por ejemplo, si consideramos el grafo **G1**, una trayectoria para ir desde el vértice 1 hacia el vértice 4 sería:

1, 3, 2, 4

y esta es una trayectoria simple. Todos los vértices en ella son diferentes.

Si en el mismo grafo **G1** consideramos la siguiente trayectoria:

1, 3, 2, 4, 1

también será una trayectoria simple. Todos los vértices son diferentes excepto el primero y el último.

Si consideramos una trayectoria como: 1, 3, **2**, 4, **2**, 1

ésta ya no es una trayectoria simple porque por el vértice 2 se pasa dos veces.

Ciclo: es una trayectoria simple en la cual el primero y el último vértice son el mismo. Por ejemplo, en el mismo grafo **G1** la trayectoria

1, 3, 2, 4, 1

es un ciclo.

Grafo Conectado: se dice que un grafo es conectado si desde cualquier vértice **i** del grafo se puede ir hacia cualquier otro vértice **j** del grafo. Este concepto se usa para grafos no dirigidos. Para grafos dirigidos se utiliza el concepto de **Grafo Fuertemente Conectado.** Se dice que un grafo dirigido está fuertemente conectado se desde cualquier vértice se puede viajar hacia cualquier otro vértice del grafo.

Para grafos no dirigidos: el máximo número de lados es: $n * (n-1)/2$

siendo **n** el número de vértices del grafo

Para grafos dirigidos: el máximo número de lados es n*(n-1).

Un grafo dirigido es **completo** cuando tiene **n** vértices y n*(n-1) lados.

13.4 REPRESENTACIÓN DE GRAFOS

En general, los grafos se pueden representar de cuatro formas:

1. Matriz de Adyacencia.
2. Listas Ligadas de Adyacencia.
3. Multilistas de Adyacencia.
4. Matriz de Incidencia.

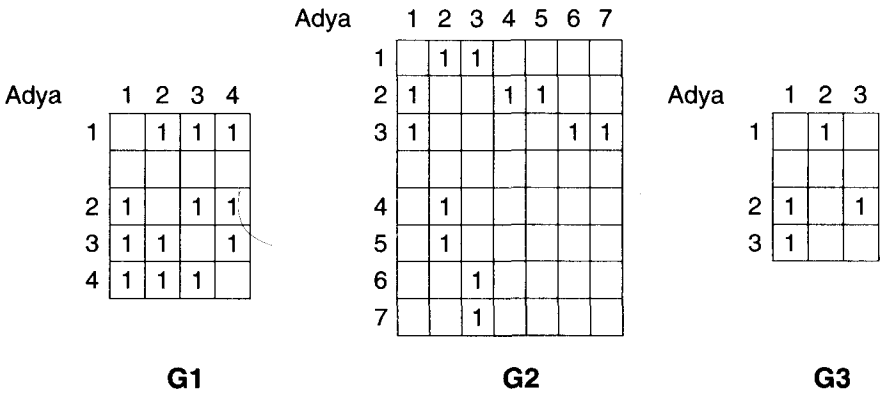
Matriz de adyacencia

Es una matriz cuadrada de orden **n**, siendo **n** el número de vértices del grafo.

Si llamamos **Adya** dicha matriz, tendremos:

$$\textbf{Adya(i,j)} = \begin{cases} 1\text{: Si existe lado } (V_i, V_j) \\ 0\text{: de lo contrario.} \end{cases}$$

La representación, como matriz de adyacencia, de los grafos **G1, G2** y **G3** es:



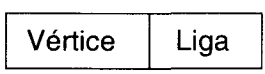
Para conocer el grado de cualquier vértice, se recorrerá la fila correspondiente a tal vértice contando los unos. El número de unos da el grado del vértice.

Para grafos no dirigidos la Matriz de adyacencia es **Simétrica** y en ocasiones **Dispersa**, por consiguiente se podrá representar como una matriz dispersa.

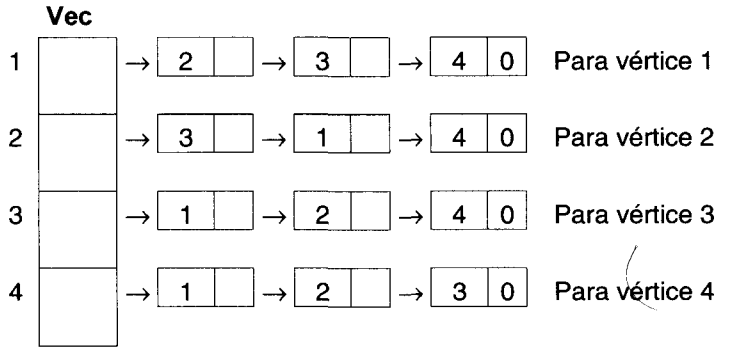
En grafos dirigidos, para conocer el grado saliente de un vértice se cuentan los unos en la fila correspondiente a ese vértice. El grado entrante se obtiene contando los unos de la columna correspondiente a ese vértice. El grado total es la suma de ambos.

Listas ligadas de adyacencia

Para representar grafos como listas ligadas de adyacencia, se tendrá una lista simplemente ligada por cada vértice que tenga el grafo. Cada lista ligada es una lista de los vértices adyacentes a él. Cada registro tendrá un vértice adyacente. La configuración del registro para dicha representación es:

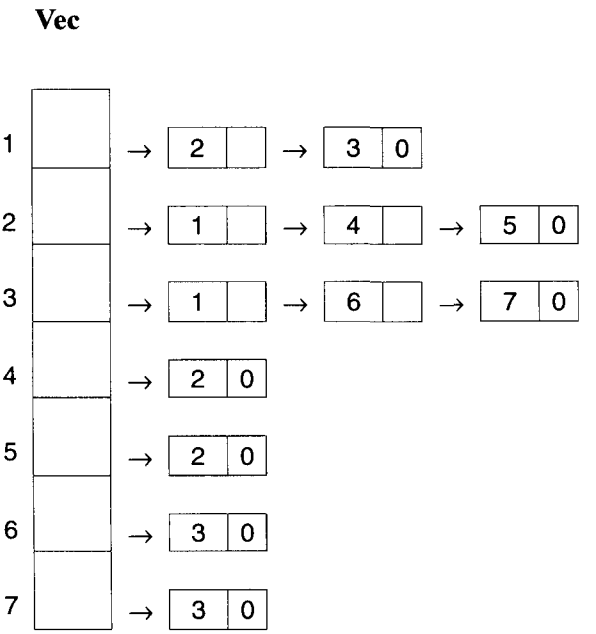


Representación del grafo G1:



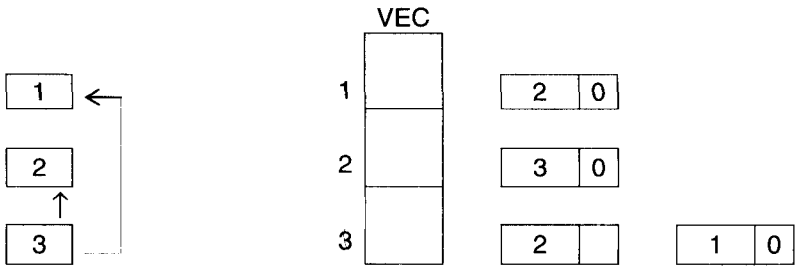
Vec(i): Apunta hacia el primer registro de la lista correspondiente al vértice **i**.

Representación del grafo G2:



Para conocer el grado de un vértice **i** se entra por la lista correspondiente al vértice **i** y se cuentan los registros existentes. Determinar el grado de un vértice es contar los registros en una lista.

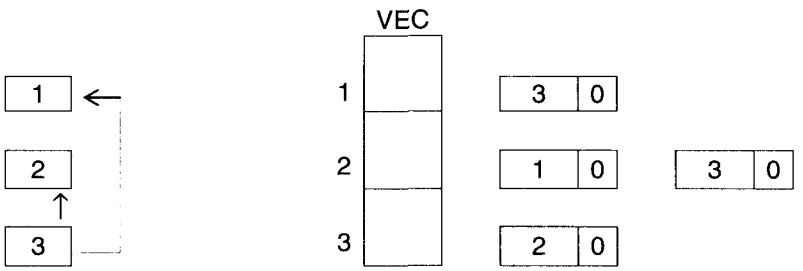
Consideremos un grafo dirigido **G4**:



La representación anterior, como lista ligada, representa realmente la relación "adyacente hacia".

En el ejemplo, el grado saliente se obtiene contando el número de registros correspondiente a cada vértice.

El grado entrante se conocerá recorriendo todas las listas y contando los registros que contienen el vértice de interés. Este procedimiento es ineficiente ya que implica recorrer todas las listas ligadas del grafo. Para evitar esto, puede construírse una lista ligada con la relación «adyacente desde».



Con la representación anterior podremos determinar el grado entrante de un vértice recorriendo únicamente la lista correspondiente a ese vértice.

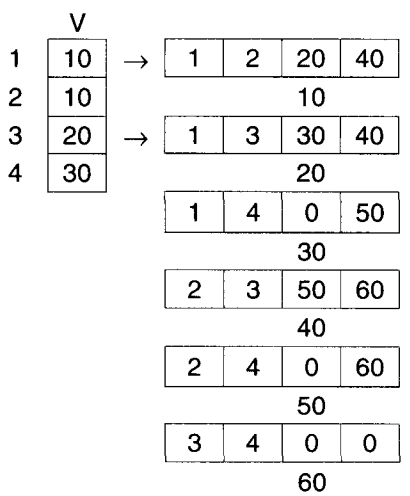
Multilistas de adyacencia

Se usará un registro para representar cada lado del grafo. Un lado está conformado por una pareja de vértices. La configuración del registro es la siguiente:

		LV_i	LV_j
V_i	V_j	Liga asociada con V_i	Liga asociada con V_j

V_i: Campo que contiene el vértice i.
V_j: Campo que contiene el vértice j.
LV_i: Apunta hacia otro registro que representa un lado incidente a Vi.
LV_j: Apunta hacia otro registro que representa un lado incidente a Vj_i

Teniendo definida la configuración del registro representemos el grafo G1 como multilista de adyacencia. Recordemos que una multilista es una lista ligada en la cual cada registro pertenece simultáneamente a más de una lista.

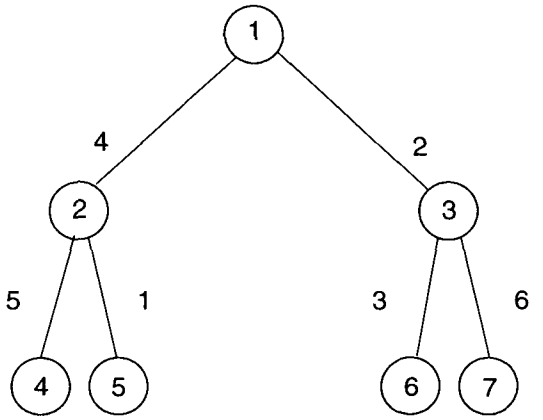


V(i): Apunta hacia el primer registro de la lista de registros que representan los lados incidentes al vértice **i**.

Matriz de incidencia

Es una matriz de **m** filas y **n** columnas siendo: m = número vértices del grafo.
n = número lados del grafo.

lo que implica que debemos numerar los lados del grafo. Dicha numeración se hace aleatoriamente. Consideremos el grafo **G2.**



Al tener numerados los lados podremos hacer la representación como matriz de incidencia. En nuestro ejemplo: **m** = 7 y **n** = 6, por tanto tendremos una matriz de 7 lados y 6 columnas

Si llamamos **Inci** dicha matriz tendremos:

$$\text{Inci}(i,j) = \begin{cases} 1\text{: Si el lado } \mathbf{j} \text{ es incidente sobre el vértice } \mathbf{i}. \\ 0\text{: De lo contrario.} \end{cases}$$

Las filas representan los vértices y las columnas, los lados.

La representación del grafo **G2** como matriz de incidencia es:

Inci	1	2	3	4	5	6
1		1		1		
2	1			1	1	
3		1	1			1
4					1	
5	1					
6			1			
7						1

Como podrá observar cada columna sólo tendrá dos unos: en las filas correspondientes a los vértices que conforman ese lado.

El grado de un vértice **i** se halla contando los unos sobre la fila **i**.

La decisión de cuál forma de representación elegir dependerá del problema que se esté tratando y del gusto del diseñador.

13.5 RECORRIDOS SOBRE GRAFOS

Recorrer un grafo se puede hacer de dos maneras: una que se llama DFS y otra que se llama BFS

DFS

Recibe este nombre porque son las iniciales de las palabras en inglés **D**epth **F**irst **S**earch, lo cual traducido literalmente es algo así como búsqueda del primero en profundidad.

El recorrido DFS consiste en: estando ubicados en un vértice **i** cualquiera, determinar los vértices adyacentes, de esos vértices adyacentes elegir uno que no haya sido visitado y a partir de ahí iniciar nuevamente recorrido DFS.

Observe que estamos planteando una definición recursiva para el recorrido.

Lo anterior implica que debemos controlar cuáles vértices han sido visitados y cuáles no, además debemos tener un mecanismo que me permita conocer cuáles son los vértices adyacentes a un vértice dado.

Para controlar cuáles vértices han sido visitados y cuáles no se usa un vector que llamaremos VISITADO que tendrá tantos elementos como vértices tenga el grafo.

$$\text{VISITADO(i)} = \begin{cases} \text{0: el vértice i no ha sido visitado} \\ \text{1: el vértice i ya fue visitado} \end{cases}$$

Inicialmente todos los elementos del vector VISITADO estarán en cero.

	1	2	3	4	5	6	7	8	9
VISITADO	0	0	0	0	0	0	0	0	0

Consideremos el siguiente grafo:

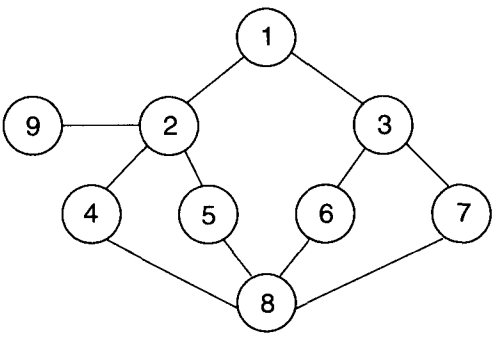


Figura 13.2

Sea **V=1** el vértice a partir del cual iniciamos el recorrido DFS:

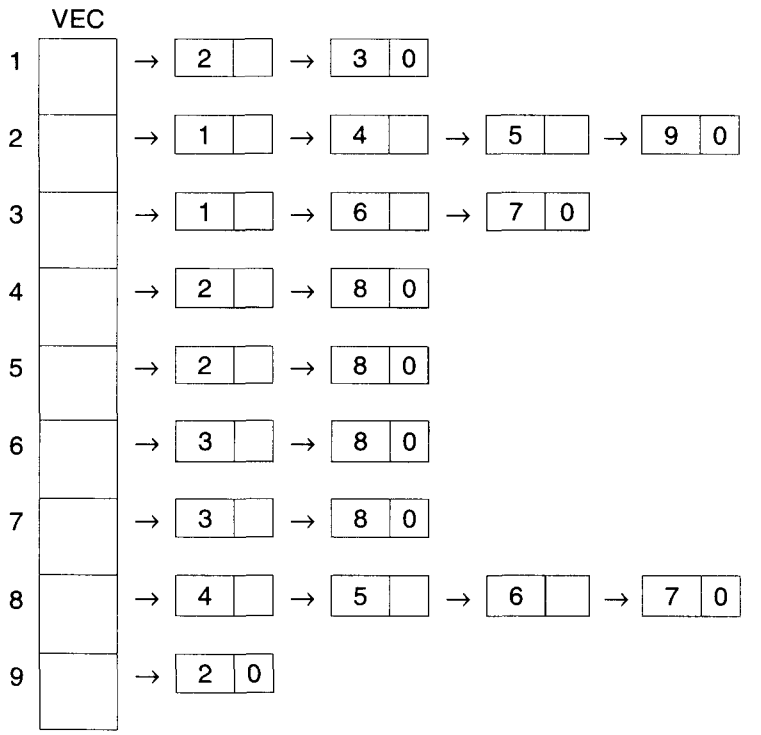
El recorrido DFS sobre dicho grafo es: 1, 2, 4, 8, 5, 6, 3, 7, 9

Un algoritmo general que efectúa dicho recorrido es:

```
sub_programa dfs(v)
        visitado(v) = 1
        para todo vértice w adyacente a v haga
                if visitado(w) = 0 then
                        dfs(w)
        end(if)
fin(para)
fin(dfs)
```

El parámetro **v** indica el vértice a partir del cual se inicia el recorrido DFS. Determinar los vértices adyacentes a un vértice dado dependerá de la forma como se tenga representado el grafo.

Desarrollemos este procedimiento general para el grafo de la figura 13.2 teniéndolo representado como listas ligadas de adyacencia.




```
sub_programa dfs(v)
      visitado(v) = 1
      p= vec(v)
      while  p <> 0  do
            w = vertice(p)
            if  visitado(w)=0  then
                  dfs(w)
            end(if)
            p=liga(p)
      end(while)
fin(dfs)
```

BFS

Recibe este nombre porque son las iniciales de las palabras en inglés **B**readth **F**irst **S**earch, las cuales traducidas literalmente significan algo así como Búsqueda del primero a lo ancho.

Hay una diferencia sustancial entre este recorrido y el DFS: mientras que en el recorrido DFS no se visitan de una vez todos los adyacentes a un vértice dado, aquí hay que visitar primero todos los vértices adyacentes a un vértice dado antes de continuar el recorrido.

Consideremos nuevamente el grafo de la figura 13.2

Sea **V=1** el vértice a partir del cual se inicia el recorrido:

Primero se visita el vértice 1.

Luego visitamos los vértices adyacentes al vértice 1. Dichos vértices son el vértice 2 y el vértice 3.

Hasta aquí nuestro recorrido es:

1, 2, 3

El recorrido continúa visitando primero los vértices adyacentes al vértice 2 y luego los vértices adyacentes al vértice 3.

Cuando visitamos los vértices adyacentes al vértice 2 nuestro recorrido es:

1, 2, 3, 9, 4, 5

Cuando visitamos los vértices adyacentes al vértice 3 nuestro recorrido es:

1, 2, 3, 9, 4, 5, 6, 7

Luego continuaremos visitando los vértices adyacentes a los vértices 9, 4, 5, 6 y 7.

Lo anterior me sugiere que debo manejar una cola que indique el orden en que he ido visitando los vértices.

Como cada vértice sólo lo debo visitar una vez también debo manejar un vector para controlar cuáles vértices he visitado y cuáles no. Este es el vector que hemos llamado VISITADO.

El algoritmo general para hacer un recorrido BFS es:

```
sub_programa bfs(v)
      visitado(v) = 1
      encolar(v)
      mientras cola no vacía haga
```

```
            desencolar(v)
            para todo vértice w adyacente a v haga
                  if  visitado(w) = 0  then
                        visitado(w) = 1
                         encolar(w)
                  end(if)
            fin(para)
      fin(mientras)
fin(bfs)
```

La determinación de los vértices **w** adyacentes a un vértice **v** dependerá de la forma como se tenga representado el grafo.

A continuación presentamos el grafo del ejemplo representado como matriz de adyacencia y un algoritmo completo para hacer el recorrido BFS teniendo la cola representada en un vector.

ADYA

	1	2	3	4	5	6	7	8	9
1		1	1						
2	1			1	1				1
3	1					1	1		
4		1						1	
5		1						1	
6			1					1	
7			1					1	
8				1	1	1	1		
9		1							

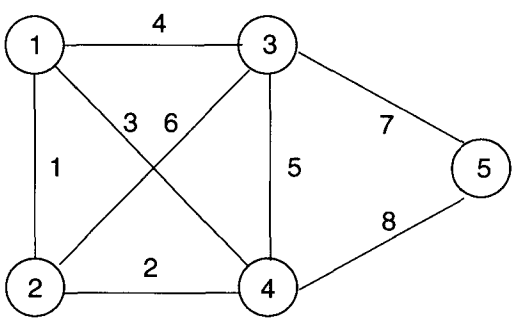
```
sub_programa bfs(v)
      visitado(v) = 1
      primero = 0
      ultimo = 1
      cola(1) = v
      while  primero <> ultimo  do
            primero = primero +1
            v = cola(primero)
            for  w = 1 to n  do
                  if  adya(v,w) = 1  then
                        if  visitado(w) = 0  then
```

```
                        visitado(w) = 1
                        ultimo = ultimo + 1
                        cola(ultimo) = w
                    end(if)
                end(if)
            end(for)
        end(while)
fin(bfs)
```

13.6 SPANNING TREE

Es un árbol resultante de recorrer un grafo. Muchos textos lo traducen como árbol expandido.

Como resulta del recorrido sobre un grafo podremos decir que hay spanning tree DFS y spanning tree BFS.



Algoritmo de Kruskal

Es un algoritmo que permite construir un spanning tree de costo mínimo. Para poder ejecutar el algoritmo de Kruskal se necesitan dos cosas:

1. Organizar los lados ascendentemente por costo. Para nuestro ejemplo:

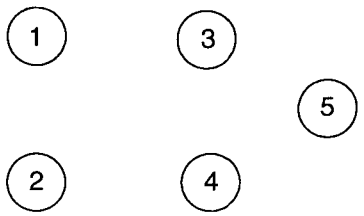
   E = {(1,2), (2,4), (1,4), (1,3), (3,4), (2,3), (3,5), (4,5)}

   y

2. definir un conjunto de conjuntos en los que cada subconjunto está inicialmente conformado por cada uno de los vértices que conforman el grafo. Para nuestro ejemplo:

   V= {{1}, {2}, {3}, {4}, {5}}

En primera instancia, la representación de **v** indica que los vértices no están conectados, entre sí. Es decir, es como si los tuviéramos así:



Para un grafo con **n** vértices, el Spanning tree tendrá **n – 1** lados.

Llamemos ST el conjunto que tendrá los lados que conforman el Spanning Tree.

```
algoritmo de kruskal
      st = {}
      i = 0
      while  i < n-1  do
            escoger de e lado (u,w) de costo mínimo
            borrar de e lado (u,w) escogido
            si lado (u,w) no forma ciclo en st
                incluír lado (u,w) en st
                unir en v los subconjuntos que contenga a u y a w
                i = i+1
            fin(si)
      end(while)
fin(algoritmo)
```

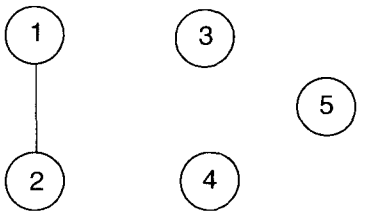
A medida que se van conectando los vértices se hace la unión de los subconjuntos de V que contengan a dichos vértices.

Se tiene un ciclo cuando los vértices a conectar están en el mismo subconjunto de V.

En nuestro ejemplo, la primera vez escogemos el lado (1,2); como los vértices 1 y 2 están en diferentes subconjuntos de V entonces incluimos el lado (1,2) en el conjunto ST y hacemos la unión de los subconjuntos de V que contienen los vértices 1 y 2.

El conjunto E quedará así: E = {(2,4), (1,4), (1,3), (3,4), (2,3), (3,5), (4,5)}
El conjunto ST quedará así: ST = {(1,2)}
El conjunto V quedará así: V = {{1,2}, {3}, {4}, {5}}

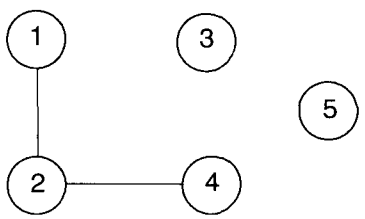
y el spanning tree, gráficamente así:



Al ejecutar el ciclo por segunda vez, escogemos el lado (2,4). Como ambos vértices están en diferentes subconjuntos de V, estos vértices no forman ciclo. Por tanto, se incluye el lado (2,4) en ST, se borra de E y se hace la unión de los subconjuntos de V que contienen los vértices 2 y 4.

El conjunto E quedará así: E = {(1,4), (1,3), (3,4), (2,3), (3,5), (4,5)}
El conjunto ST quedará así: ST = {(1,2), (2,4)}
El conjunto V quedará así: V = {{1,2,4}, {3}, {5}}

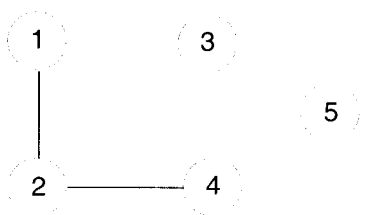
y el spanning tree, gráficamente así:



Luego escogemos el lado (1,4) y las cosas quedan así:

El conjunto E quedará así: E = {(1,3), (3,4), (2,3), (3,5), (4,5)}
El conjunto ST quedará así: ST = {(1,2), (2,4)}
El conjunto V quedará así: V = {{1,2,4}, {3}, {5}}

y el spanning tree, gráficamente así:



Es decir, el lado (1,4) se desechó porque los vértices 1 y 4 estaban en un mismo subconjunto de V y por tanto conformaban ciclo.

Se continúa con el proceso descrito hasta completar n – 1 lados en el conjunto ST.

Las instrucciones apropiadas se escribirán de acuerdo a la forma como el lector quiera representar los conjuntos

13.7 DETERMINACIÓN DE DISTANCIAS Y RUTAS MÍNIMAS (ALGORITMO DE DIJKSTRA)

Un grafo dirigido se puede usar para representar una red vial, considerando los vértices como ciudades y los lados como los kilómetros de distancia entre dos ciudades, o el tiempo para ir desde una ciudad hacia otra, o el costo para viajar desde una ciudad hacia otra.

Consideremos el grafo de la figura 13.3.

Para un turista que llegue, digamos al vértice 1, le interesará conocer dos cosas: una, conocer hacia qué vértices puede viajar, y dos, hacia aquellos vértices a los que puede viajar determinar cuál es la menor distancia para llegar a ellos.

Consideremos la representación en matriz del grafo anterior: La representación es similar a la de una matriz de adyacencia. La diferencia estriba en que en vez de un 1 en la posición **Dist(i,j)** colocaremos la distancia que hay entre los vértices **i** y **j**.

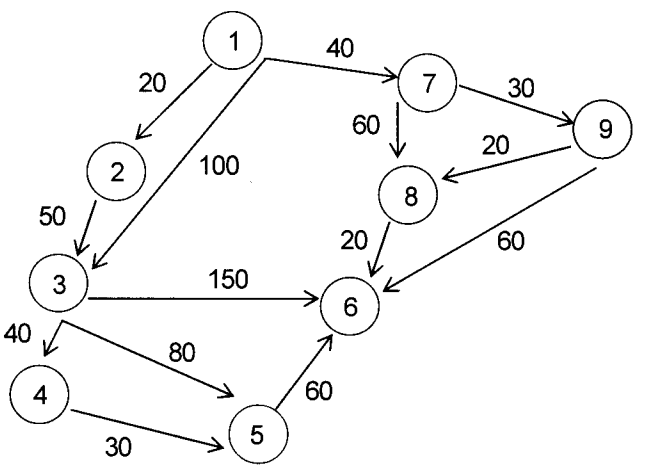


Figura 13.3

Para los vértices que no están conectados colocaremos ∞, indicando que no hay conexión directa entre esos vértices, es decir, que ese lado no existe.

Las posiciones correspondientes a la diagonal principal serán cero, es decir la distancia para viajar desde el vértice **i** hacia el vértice **i** es cero.

La matriz **Dist** de nuestro grafo queda así:

DIST

	1	2	3	4	5	6	7	8	9
1	0	20	100					40	
2		0	50						
3			0	40	80	150			
4				0	30				
5					0	60			
6						0			
7							0	60	30
8						20		0	
9						60		20	0

Las posiciones en blanco representan ∞.

El algoritmo utiliza un vector llamado **Dist_min**, el cual retornará las distancias mínimas para viajar desde un vértice **V** hacia los demás vértices. Aquellas posiciones **j** que regresen siendo ∞ significarán que no hay ruta para viajar desde **V** hacia ese vértice **j**.

Inicialmente el vector **Dist_min** tendrá los datos correspondientes al vértice V, siendo V el vértice de partida.

En nuestro ejemplo consideraremos V = 1.

La configuración inicial del vector **Dist_min** es la siguiente:

	1	2	3	4	5	6	7	8	9
Dist_min	0	20	100					40	

Adicionalmente, requerimos de un vector que indique hacia qué vértice ya se ha calculado la distancia mínima. Llamemos S dicho vector.

Inicialmente el vector S tendrá cero en todas sus posiciones, indicando que hacia ningún vértice se ha calculado aún la distancia mínima.

	1	2	3	4	5	6	7	8	9
S	0	0	0	0	0	0	0	0	0

En general:

S(i) = 0: Hacia el vértice I no se ha calculado la menor distancia.

1: Hacia el vértice I ya se calculó la menor distancia.

El algoritmo de DIJKSTRA es el siguiente:

```
1     sub_programa minimas_distancias(dist, n, v, dist_min)
2           for  i = 1 to n  do
3                  dist_min(i) = dist(v,i)
4                  s(i)=0
5           end(for)
6           s(v) = 1
7           i = 1
8           while  i < n – 1   do
9                 escoger vértice w tal que dist_min(w) sea la menor
10                  de los elementos de dist_min cuyo correspondiente
11                  elemento de s es cero.
12                s(w) = 1
13                i = i+1
14                para todo vértice j con s(j)=0 haga
15                       dist_min(j) = menor(dist_min(j),dist_min(w)+dist(w,j)
16                fin(para)
17           end(while)
18    fin(minimas_distancias)
```

Expliquemos algunas instrucciones de dicho algoritmo.

Empecemos con las instrucciones de la 9 a la 11.

Supongamos que el vector **Dist_min** tiene los siguientes datos en un momento dado:

	1	2	3	4	5	6	7	8	9
Dist-min	0	20	100				40		

y que el vector S está así:

	1	2	3	4	5	6	7	8	9
S	1	1	0	0	0	0	0	0	0

El vértice W a escoger es W = 7.

No se escoge el vértice 2 porque su correspondiente valor en S es cero.

Veamos ahora las instrucciones de la 14 a la 16.

Para un vértice cualquiera, **Dist_min(j)** da la distancia mínima para viajar desde el vértice de partida **V** hacia el vértice **j**.

Lo que debemos tener en cuenta es si esta distancia es la definitiva o si es todavía una aproximación.

Esto es lo que nos informa S(J). Si S(J) es 1 es porque es la distancia mínima definitiva; si S(J) es cero significa que aún no se ha calculado la distancia mínima definitiva para llegar al vértice J. Puede suceder que pasando por otro vértice, digamos W, se obtenga una distancia menor.

Esquemáticamente podemos sintetizar así:

DIST_MIN(J): distancia para viajar desde V hacia J.
DIST_MIN(W): distancia para viajar desde V hacia W.
DIST(W, J): distancia para ir **directamente** desde W hacia J.
DIST_MIN(W) + DIST(W,J): distancia para ir desde V hacia J pasando por el vértice W.

Instrucciones 14 a 16 lo que hacen es examinar si para llegar al vértice J existe una distancia menor pasando por un vértice W previamente escogido que la distancia que ya se había calculado. Esto lo hace únicamente para aquellos vértices con S(J) = 0, o sea aquellos a los cuales aún no se les ha determinado la distancia mínima.

Para el grafo del ejemplo con V = 1 se obtiene el siguiente vector de distancias mínimas:

	1	2	3	4	5	6	7	8	9
DIST-MIN	0	20	70	110	140	110	40	90	70

Hasta aquí sólamente hemos calculado las distancias menores, nos interesará conocer la ruta que hay que seguir para obtener esta distancia mínima.

Para ello se usa un vector adicional llamado RUTA.

Inicialmente a **Ruta(i) = i** indicando que para llegar al vértice **i** se hace directamente desde el vértice de partida **V**.

	1	2	3	4	5	6	7	8	9
RUTA	1	2	3	4	5	6	7	8	9

La modificación de este vector se hará en las instrucciones 14 a 16.

Cuando encontremos que para llegar a un vértice J es más corto pasando por el vértice W le asignaremos W a **Ruta(j)**.

Al final tendremos un vector de **Ruta** conformado, digamos así:

	1	2	3	4	5	6	7	8	9
RUTA	1	2	2	3	5	4	7	6	9

Si consideramos la posición 8 del vector, allí encontramos un 6. Esto significa que para llegar al vértice 8 hay que pasar por el vértice 6.

En la posición 6 encontramos un 4. Esto significa que para llegar al vértice 6 hay que pasar por el vértice 4.

En la posición 4 encontramos un 3. Esto significa que para llegar al vértice 4 hay que pasar por el vértice 3.

En la posición 3 se encuentra un 2. Esto significa que para llegar al vértice 3 hay que pasar por el vértice 2.

En la posición 2 se encuentra un dos. Esto significa que al vértice 2 se llega directamente desde el vértice V.

Resumiendo, cuando **Ruta(i) = i**, al vértice **i** se llega directamente desde el vértice **V**. Si **Ruta(i) <> i** significa que para llegar al vértice **i** desde el vértice **V**, hay que pasar por el vértice **Ruta(i)**.

A continuación presentamos el algoritmo de DIJKSTRA modificado para construir el vector **Ruta**.

```
sub_programa minimas_distancias(dist, n, v, dist_min)
      for i = 1 to n do
            dist_min(i) = dist(v,i)
            s(i)=0
      ruta(i) = i
      end(for)
      s(v)=1
      i=1
      while i < n – 1 do
                escoger vértice w tal que dist_min(w) sea la menor
                  de los elementos de dist_min cuyo correspondiente
                  elemento de S es cero.
                s(w) = 1
                i = i+1
                for j = 1 to n do
                          if s(j) = 0 then
                                paso = dist_min(w) + dist(w,j)
                                if paso < dist_min(j) then
                                      dist_min(j) = paso
                                      ruta(j) = w
                                end(if)
                          end(if)
                end(for)
      end(while)
end(sub_programa)
```

Ocupémonos ahora de la primera pregunta del turista. Hacia cuáles vértices puede viajar.

Para resolver este interrogante construyamos la matriz de adyacencia que representa el grafo. Llamemos **A** esta matriz.

A	1	2	3	4	5	6	7	8	9
1		1	1				1		
2			1						
3				1	1	1			
4					1				
5						1			
6									
7								1	1
8						1			
9						1		1	

Para resolver este interrogante basta con construir el cierre transitivo de la matriz A. Llamemos esta matriz la matriz A+.

$$A+(i,j) = \begin{cases} 0: \text{No se puede viajar desde el vértice } \mathbf{i} \text{ hacia el vértice } \mathbf{j}. \\ 1: \text{Si se puede viajar desde el vértice } \mathbf{i} \text{ hacia el vértice } \mathbf{j}. \end{cases}$$

EJERCICIOS PROPUESTOS

1. Escriba algoritmos para recorrer en DFS grafos representados en alguna de las formas definidas en el texto.

2. Escriba algoritmos para recorrer en BFS grafos representados en alguna de las formas definidas en el texto.

3. Elabore algoritmos para determinar si un grafo no dirigido tiene ciclos o no. Considere todas las formas de representación.

4. Elabore algoritmos para imprimir todas las trayectorias simples para viajar desde un vértice V hacia un vértice W en un grafo dirigido.

5. Elabore un algoritmo que construya una matriz que indique hacia qué vértice se puede viajar desde un vértice dado hacia los demás vértices. Es decir, un algoritmo que construya la matriz A+.

6. Elabore un algoritmo que procese el vector de ruta creado en el algoritmo de Dijkstra y que imprima el camino correspondiente.

7. Elabore algoritmos para construir representaciones de grafos con base en otras representaciones dadas. Considere todas las posibilidades.

14

MANEJO DE CARACTERES (HILERAS O STRINGS)

14.1 DEFINICIÓN

Una hilera es un objeto de datos compuesto, construido con base en el tipo primitivo char.

En los primeros lenguajes de programación las hileras aparecían sólo como constantes que se utilizaban únicamente como entrada y salida. No había operaciones sobre ellas. A medida que se desarrollaron los lenguajes de programación se presentó la necesidad de construir reconocedores de léxico, interpretación de hileras, modificaciones, y en general operaciones sobre ellas, máxime cuando comenzaron a aparecer los procesadores de texto.

Aquí veremos diferentes formas de representar hileras y cómo manipularlas.

Para ser un poco formales, comencemos definiendo la estructura hilera en abstracto con algunas de las operaciones que se implementan.

```
ESTRUCTURA HILERA
funciones
        Nula() → hilera
        Añadir(hilera, carácter) → hilera
        Esnula(hilera) → lógico
        Longitud(hilera) → entero
        Concat(hilera, hilera) → hilera
        Subhilera(hilera, entero, entero) → hilera
        Posicion(hilera, hilera) → entero
Axiomas
        Para todo  S, T ∈  hilera
                   i, j ∈  entero
```

```
        c, d ∈ carácter

Esnula(Nula) ::= verdad
Esnula(Añadir(S, c) ::= falso

Longitud(Nula) ::= 0
Longitud(Añadir(S, c) ::= 1 + Longitud(S)

Concat(S, Nula) ::= S
Concat(S, Añadir(T, c) ::= Añadir(Concat(S, T), c)

Subhilera(Nula, i, j) ::= Nula
Subhilera(Añadir(S, c), i, j) ::=
    IF j = 0 OR i+j-1 > Longitud(Añadir(S, c)) THEN
        Nula
    ELSE
      IF i+j-1 = Longitud(Añadir(S, c)) THEN
        Añadir(Subhilera(S, i,j-1), c)
      ELSE
        Subhilera(S, i, j)

Posicion(S, Nula) ::= Longitud(S) + 1
Posicion(Nula, Añadir(T, d)) ::= 0
Posicion(Añadir(S, c),Añadir(T, d)) ::=
    IF c = d AND Posicion(S, T) = Longitud(S) – Longitud(T) + 1
        Posicion(S, T)
    ELSE
        Posicion(S, Añadir(T, d))
```

Las funciones definidas anteriormente son:

Nula: crea una hilera vacía.
Añadir: inserta un carácter al final de una hilera.
Esnula: determina si una hilera es vacía o no.
Longitud: calcula el número de caracteres de una hilera.
Subhilera: retorna una hilera de **j** caracteres consecutivos a partir de la posición **i** de una hilera dada.
Concat: añade la segunda hilera dada a continuación de la primera.
Posicion: retorna la posición a partir del cual encuentra la segunda hilera en la primera, en caso de no encontrarla retorna cero.

14.2 OPERACIONES CON HILERAS

Consideremos ahora las operaciones básicas que se pueden ejecutar con hileras:

Asignación

En el lado izquierdo tendremos el nombre de la variable a la cual se le asigna una hilera dada. En el lado derecho se tendrá, o una hilera o una variable tipo hilera. por ejemplo: si S, T, U y V son variables tipo hilera podemos tener las siguientes instrucciones:

```
S = 'abc'
T = 'def'
U = T
V = ''
```

En el primer ejemplo la variable S contendrá la hilera 'abc'.
En el segundo ejemplo la variable T contendrá la hilera 'def'
En el tercer ejemplo la variable U contendrá la hilera 'def'
En el cuarto ejemplo la variable V contendrá la hilera vacía.

Función subhilera

Forma general: Sub(S, i, j) con $1 \; d \leq i \; d \leq i+j-1 \; d \leq n$

siendo **n** la longitud de la hilera S

Esta función, a partir de la posición **i** de la hilera **S** extrae **j** caracteres creando una nueva hilera y dejando intacta la hilera original **S**. Por ejemplo, si tenemos **S** = 'mazamorra' y damos la instrucción:

```
T = Sub(S, 4, 5)
```

La variable T quedará conteniendo 'amorr'.

Función longitud

Forma general: longitud(S)

Esta función retorna el número de caracteres que tiene la hilera **S**. Por ejemplo, si tenemos S = 'mazamorra' y damos la instrucción:

```
n = longitud(S)
```

La variable **n** quedará valiendo 9.

Función concatenar:

Forma general: Concat(S, T)

Esta función inserta la hilera T a continuación del último carácter de la hilera S. Por ejemplo, si S = 'nacio' y T = 'nal' y damos la instrucción:

```
U = Concat(S, T)
```

La variable U quedará conteniendo la hilera 'nacional'.

Función posición:

Forma general: Pos(S, T)

Esta función determina a partir de cuál posición de la hilera T se encuentra la hilera S, si es que se encuentra en la hilera T. En caso de no encontrar la hilera T en la hilera S retornará cero. Por ejemplo, si T = 'mazamorra' y S = 'amor' y damos la instrucción:

```
M = Pos(S, T)
```

la variable **m** quedará valiendo 4.

Las siguientes funciones las definiremos por facilidad, ya que todas se pueden elaborar en función de las anteriormente definidas.

Procedimiento insertar:

Forma general: Insert(S, i, T)

Este procedimiento inserta la hilera T a continuación del carácter de la posición **i** de la hilera S. Por ejemplo, si tenemos S = 'nal' y T = 'ciona' y damos la instrucción:

```
Insert(S,2,T)
```

la hilera S quedará valiendo 'nacional'

Es bueno resaltar que en este caso la hilera S queda modificada mientras que la hilera T permanece intacta.

Procedimiento borrar:

Forma general: Borrar(S, i, j)

Este procedimiento, a partir de la posición **i** de la hilera S borra **j** caracteres de la hilera S. Por ejemplo, si S='amotinar' y damos la instrucción:

```
Borrar(S, 4, 4)
```

la hilera S queda valiendo 'amor'

Es bueno anotar también aquí, que la hilera S queda modificada.

Procedimiento reemplazar:

Forma general: Replace(S, i, j, T)

Este procedimiento: a partir de la posición **i** de la hilera S reemplaza **j** caracteres por la hilera T. Por ejemplo, si la hilera S = 'abcdef' y la hilera T = 'xyz' y damos la instrucción:

```
Replace(S, 3, 2, T)
```

la hilera S queda valiendo 'abxyzef'

Este procedimiento modifica la hilera S mientras que la hilera T permanece intacta.

Las operaciones dadas anteriormente nos proporcionan las facilidades para manipular hileras en cualquier lenguaje de programación.

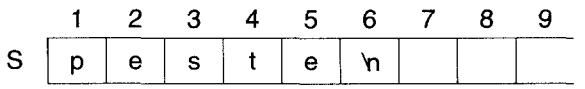
14.3 REPRESENTACIÓN DE HILERAS

Cuando uno decide cómo va a representar un objeto dado se debe tener en cuenta el costo de las operaciones que se desean ejecutar sobre ese objeto. Además, se debe tener en cuenta el consumo de memoria de la representación escogida.

Para representar hileras existen básicamente dos formas: secuencialmente o como listas ligadas.

Secuencialmente

En esta representación los caracteres de una hilera se colocan consecutivamente, digamos que en un vector. Si tenemos la hilera S = 'peste', sólo necesitamos una variable que indique a partir de qué posición del vector se almacena dicha hilera. El final de la hilera lo podremos controlar con algún símbolo especial.



Con esta forma de representación algunas funciones serán fácil y eficientemente implementables. Funciones como Longitud, Subhilera, Concatenacion tendrán algoritmos fáciles y eficientes.

Para hileras muy cortas se desperdicia memoria, ya que no se utiliza la totalidad del vector definido.

Además, operaciones como inserciones y borrados tendrán algoritmos no muy eficientes.

Consideremos la operación o función **Posición** teniendo la hilera representada en forma secuencial.

Un algoritmo que determine **POS** es el siguiente:

```
function pos(p, t)
    m = longitud(p)
    n = longitud(t)
    i = 1
    j = 1
    while i <= m and j <= n do
        if p(i) = t(j) then
            i = i + 1
            j = j + 1
            if i > m then
                return(j – i + 1)
            end(if)
        else
            j = j – i + 2
            i = 1
        end(if)
```

```
        end(while)
        return(0)
fin(pos)
```

Este algoritmo tiene el inconveniente de que se devuelve en la lista **T** cuando encuentra una desigualdad entre **P(i)** y **T(j)**.

Para evitar este retroceso se puede construir un vector, que llamaremos **Siguientes** el cual contendrá en la posición **i** el carácter de la hilera **P** con el cual se debe realizar la siguiente comparación para no tener que devolvernos sobre la hilera **T**.

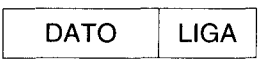
Es decir, si **Siguientes(6) = 4** significa que cuando se encuentre una desigualdad entre **Siguientes(6)** y algún **T(J)** el carácter **T(J)** habrá que compararlo con el carácter que indique el contenido de **Siguientes(6)** si éste es mayor que cero; si **Siguientes(6)** es igual a cero se continuará comparando **T(J+1)** con **P(1)**.

Teniendo construido el vector **Siguientes** el algoritmo de **POS** quedará así:

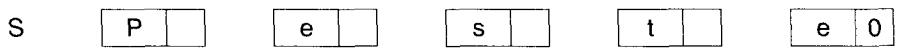
```
function pos(p, t)
        m = longitud(p)
        n = longitud(t)
        i = 1
        j = 1
        while  i <= m  and  j <= n  do
                if  p(i) = t(j)  then
                        i = i + 1
                        j = j + 1
                        if  i > m  then
                                return(j – i + 1)
                        end(if)
                else
                        if  siguientes(i) > 0  then
                                i = siguientes(i)
                        else
                                i = 1
                                j = j + 1
                        end(if)
                end(if)
        end(while)
        return(0)
fin(pos)
```

Como listas ligadas

Como siempre hemos dicho, lo primero que debemos hacer es definir la configuración del registro. Para nuestro caso sólo necesitamos dos campos: una para el carácter y otro para la liga.



Si queremos representar la hilera 'peste' tendríamos:



Con esta representación no tendríamos desperdicio de memoria y las operaciones de inserción y borrado resultarían con algoritmos eficientes.

Como ejemplos de algoritmos con la representación de listas ligadas consideremos dos casos:

La función u operación de concatenar dos hileras:

```
function concat(s, t)
        conseguir_registro(x)
        z = x
        ultimo = x
        copiar_lista(s, ultimo)
        copiar_lista(t, ultimo)
        liga(ultimo) = 0
        concat = liga(z)
        devolver_registro(z)
fin(concat)
```

el subprograma copiar_lista es:

```
sub_programa copiar_lista(L, u)
        p = L
        while  p <> 0  do
                conseguir_registro(x)
                dato(x) = dato(p)
                liga(u) = x
                u = x
                p = liga(p)
        end(while)
fin(copiar_lista)
```

El anterior algoritmo concatena la lista **S** con la lista **T** creando una nueva lista y dejando intactas las listas **S** y **T**.

Consideremos ahora otro algoritmo que inserta la hilera **T** a continuación del carácter que ocupa la posición **i** de la hilera **S**.

En nuestro algoritmo la lista **T** desaparece.

```
sub_programa insert(s, i, t)
      n = longitud(s)
      casos
            :i < 0:  write('error')
                     stop
            :i > n:  write('error')
                     stop
            :t = 0:  return
            :s = 0:  s = t
                     return
      fin(casos)
      p = s
      j = 1
      while  j < i  do
            p = liga(p)
            j = j + 1
      end(while)
      if  i = 0  then
            q = s
            p = t
      else
            q = liga(p)
            liga(p) = t
      end(if)
      while  liga(p) <> 0  do
            p = liga(p)
      end(while)
      liga(p) = q
      t = 0
fin(insert)
```

Pasemos a considerar ejemplos de aplicación con las funciones anteriormente definidas.

Empecemos considerando un ejercicio en el cual nos solicitan determinar cuál es la frecuencia de cada una de las letras del alfabeto español en un texto dado, es

decir, cuántas veces aparece la 'a', cuántas veces aparece la 'b' y así sucesivamente.

Llamemos **texto** la variable en la cual hay que hacer dicha evaluación.

Para desarrollar dicho algoritmo utilicemos una variable llamada **alfabeto**, la cual es una variable tipo hilera y que definiremos así:

alfabeto = 'abcdefghijklmnñopqrstuvwxyz'

Utilizaremos un vector de 27 posiciones, que llamaremos **frecuencia**, en cada una de las cuales guardaremos el número de veces que se encuentre alguna letra del alfabeto definido. Es decir, a la letra A le corresponde la posición 1 del vector de frecuencias, a la letra B la posición 2, a la letra C la posición 3 y así sucesivamente.

Nuestro algoritmo consistirá en recorrer los caracteres de la variable **texto**, buscando cada carácter en la variable **alfabeto** utilizando la función **pos**, cuando la encuentre sumaremos 1 a la posición correspondiente en el vector **frecuencia**.

Nuestro algoritmo es:

```
sub_programa determina_frecuencia_letras(texto)
      alfabeto = 'abcdefghijklmnñopqrstuvwxyz'
      m = longitud(alfabeto)
      for  i = to m  do
            frecuencia(i) = 0
      end(for)
      n = longitud(texto)
      for  i = 1 to n  do
            j = pos(texto(i), alfabeto)
            if  j <> 0  then
                  frecuencia(j) = frecuencia(j) + 1
            end(if)
      end(for)
      for  i = 1 to m  do
            letra = sub(alfabeto, i, 1)
            write(letra, frecuencia(i))
      end(for)
fin(determina_frecuencia_letras)
```

Consideremos ahora un algoritmo en el cual nos interesa determinar la frecuencia de cada palabra que aparezca en un texto.

Para desarrollar dicho algoritmo debemos, primero que todo, identificar cada una de las palabras del texto. Para lograr esta identificación hay que definir cuáles caracteres se utilizan como separadores de palabras. Estos caracteres pueden ser cualquier símbolo diferente de letra, es decir, la coma, el punto, el punto y coma, los dos puntos, el espacio en blanco, etc.

Nuestro algoritmo manejará las siguientes variables:

texto: variable en la cual se halla almacenado el texto a procesar.

palabra: variable tipo vector en la cual almacenaremos cada palabra que se identifique en el texto.

frecuencia: variable tipo vector en la cual almacenaremos el número de veces que se encuentre cada palabra del texto. A una palabra que se encuentre en la posición **i** del vector **palabra**, en la posición **i** del vector **frecuencia** se hallará el número de veces que se ha encontrado.

Palabra_hallada: variable en la cual almacenaremos cada palabra que se identifique en el texto.

k: variable para contar cuántas palabras diferentes hay en el texto.

n: variable que guarda el número de caracteres en el texto.

Nuestro algoritmo es el siguiente:

```
sub_programa determina_frecuencia_palabras(texto)
1.    k = 1
2.    for  i = 1 to 1000  do
3.          frecuencia(i) = 0
4.    end(for)
5.    n = longitud(texto)
6.    i = 1
7.    while  i <= n  do
8.          car = sub(texto, i, 1)
9.          p = pos(car, alfabeto)
10.         while  i < n   and   p = 0   do
11.               i = i + 1
12.               car = sub(texto, i, 1)
13.               p = pos(car, alfabeto)
14.         end(while)
```

```
15.         j = i
16.         while  i < n   and   p <> 0   do
17.                 i = i + 1
18.                 car = sub(texto, i, 1)
19.                 p = pos(car, alfabeto)
20.         end(while)
21.           palabra_hallada = sub(texto, j, i – j)
22.         palabra(k) = palabra_hallada
23.         j = 1
24.         while  palabra(j) <> palabra_hallada  do
25.                 j = j + 1
26.         end(while)
27.         if  j = k  then
28.                 frecuencia(i) = 1
29.                 k = k + 1
30.         else
31.                 frecuencia(j) = frecuencia(j) + 1
32.         end(if)
33.   end(while)
34.   k = k - 1
35.   for  i = 1 to k  do
36.         write(palabra(i), frecuencia(i))
37.   end(for)
fin(determina_frecuencia_palabras)
```

Planteemos ahora un algoritmo que considero interesante: reemplazar una palabra por otra en un texto dado.

Utilizaremos tres variables:

texto: Variable que contiene el texto donde hay que hacer el reemplazo.
vieja: Variable que contiene la hilera que hay que reemplazar.
nueva: variable que contiene la hilera que reemplazará la hilera **vieja**.

Para entender el desarrollo de este algoritmo debemos entender lo siguiente:
Si tenemos un texto T = 'aabcbabcde' y S = 'abc' y ejecutamos la instrucción

```
m = pos(S,T)
```

La variable **m** queda valiendo 2.

Esto significa que a partir de la posición 2 de la hilera T se encontró la hilera S.

Si ejecutamos la instrucción:

m = pos(S, sub(T, 5, 6)) **(fórmula 1)**

La variable **m** también queda valiendo 2. Por qué?
porque la hilera **S** la buscará en la hilera SUB(T,5,6) y esta hilera es: 'babcde'

La pregunta es: está realmente la hilera S en la posición 2 de T?. La respuesta es no.

Si planteamos una utilización de la función POS como en la fórmula 1 y queremos determinar en cuál posición de T comienza la subhilera S, al resultado obtenido habrá que sumarle **i-1** siendo **i** el segundo parámetro de la función subhilera. En nuestro caso **i** = 5.

Llamemos **pos_ini** la variable a partir de la cual se inicia la búsqueda de una hilera en una subhilera obtenida con la función SUB.

Nuestro algoritmo es:

```
sub_programa reemplaza_vieja_por_nueva(texto, vieja, nueva)
      v = longitud(vieja)
      n = longitud(nueva)
      t = longitud(texto)
      pos_ini = 1
      sw = 1
      while sw = 1 do
            p = sub(texto, pos_ini, t+1-pos_ini)
            pos_vieja = pos(vieja,p) + pos_ini - 1
            if pos_vieja > 0 then
                  replace(texto, pos_vieja, v, nueva)
                  pos_ini = pos_vieja + n + 1
                  t = longitud(texto)
                  if pos_ini > 1 + t + v then
                        sw = 0
                  end(if)
            else
                  sw = 0
            end(if)
      end(while)
end(sub_programa)
```

EJERCICIOS PROPUESTOS

1. Elabore algoritmos que ejecuten las operaciones sobre hileras teniéndolas representadas como vectores.

2. Elabore algoritmos que ejecuten las operaciones sobre hileras teniéndolas representadas como listas ligadas. Debe desarrollar algoritmos para cada una de las representaciones posibles, es decir, simplemente ligadas, simplemente ligadas circulares, simplemente ligadas circulares con registro cabeza, doblemente ligadas, etc.

3. Elabore algoritmos para determinar si una lista ligada es un palíndromo. considere todas las posibilidades de representación.

4. Elabore un algoritmo que ajuste a la derecha un texto dado.

15

PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

15.1 INTRODUCCIÓN

La programación orientada a objetos es realmente un nuevo estilo de programación, el cual, básicamente consiste en definir *clases* y poner dichas clases a comunicarse o conversar entre sí.

Una *clase* es un *tipo*, la cual tiene asociada las operaciones que se pueden ejecutar con objetos de ese tipo.

Tipo se refiere a la clase de datos que se pueden almacenar en una variable en algún lenguaje de programación. Es así, como se pueden definir variables de tipo entero, real, carácter, etc.

Cuando se define una variable de tipo entero (Ej. **int a,b,c**) significa que en esa variable sólo se podrán almacenar datos numéricos enteros y no más. Sucede además que cuando trabajamos con variables de tipo entero podemos efectuar una serie de operaciones con ellas: sumar, restar, multiplicar y dividir, y somos usuarios de dichas operaciones. Cuando escribimos **c = a + b**, la máquina o el compilador ejecuta el algoritmo de sumar, produce un resultado y se lo asigna a **c**, y no nos preocupamos de cómo es el algoritmo de sumar.

Viéndolo de esta forma, podemos deducir que hablamos de la *clase* entero, es decir, un *tipo* que tiene asociadas ciertas operaciones, de las cuales no nos preocupamos por sus algoritmos. Esta característica se denomina *encapsulamiento*, el cual consiste en hacer uso de las operaciones inherentes a una clase sin preocuparnos cómo lo hace.

En nuestro ejemplo (**int a,b,c**) definimos **a**, **b** y **c** como objetos de la clase entero (**int**), por consiguiente decimos que **a**, **b** y **c** son instancias de la clase entero. Un objeto es, entonces, una instancia de una clase.

Cuando definimos una clase, ésta consistirá de un conjunto de datos que serán privados y un conjunto de operaciones que podremos efectuar con objetos de esa clase. Dichas operaciones las seguiremos llamando *métodos*.

Consideremos como ejemplo, que vamos a definir la clase *vector*.

Para la clase vector tendremos tres datos que serán privados:

V: un arreglo de una dimensión,
n: el tamaño de dicho arreglo,
m: variable que indica cuántos datos hay en el vector **V**.

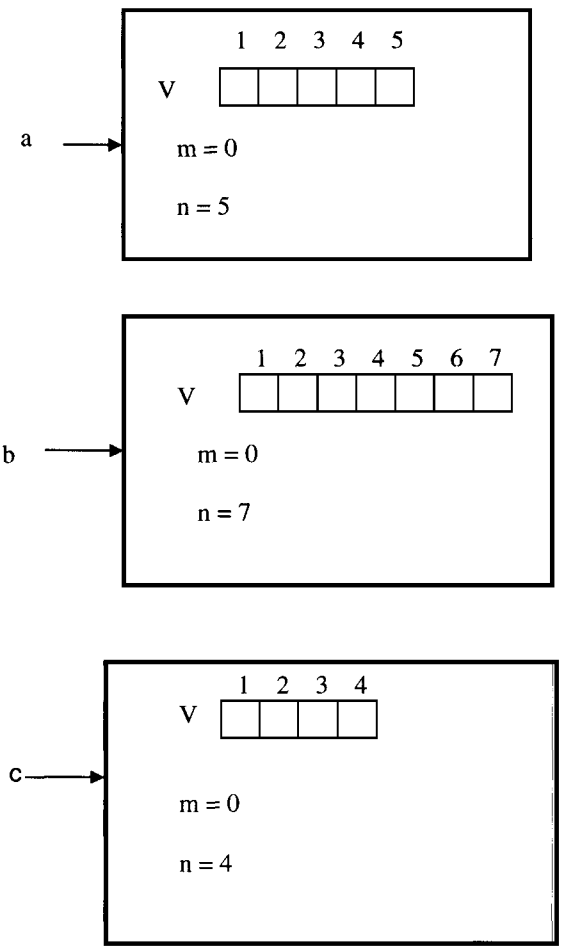
También definimos un conjunto de métodos:

- Un constructor: un método que crea un objeto vacío de la clase vector.
- Un método para insertar un dato al final del vector.
- Un método para insertar un dato al principio del vector.
- Un método para insertar un dato en una posición específica.
- Un método para buscar dónde insertar un dato en un vector ordenado.
- Métodos para ordenar los datos del vector: ascendente y descendentemente.
- Un método para borrar un dato del vector.
- Un método para imprimir los datos del vector.
- Un método para determinar si el vector está vacío o no.
- Un método para determinar si el vector está lleno o no.
- Un método que retorne la posición en la cual se halla un dato en el vector.
- Un método que retorne el dato correspondiente a una posición dada.
- Un método que retorne el número de elementos en el vector.
- Un método para actualizar el número de elementos del vector.
- Un método para destruir un objeto de la clase vector (Destructor).

Si tenemos definida una clase vector podremos definir en nuestros programas objetos de dicha clase: **Vector** a(5), b(7), c(4)

Lo cual implica que tendremos tres objetos de dicha clase, los cuales denominamos **a**, **b** y **c.**

Gráficamente la situación es la siguiente:



Definamos ahora de una manera más formal la *clase* ***Vector***:

```
Clase Vector
  Privado:
    Entero:  m, n, V(n)
  Público:
    Vector(n)          // constructor
    ins_al_final(d)
    ins_al_principio(d)
    ins_en_posición(i,d)
    asigna_dato(i,d)
    borra(d)
    ordena_asc()
    ordena_desc()
```

```
    imprime()
    asigna_m(i)
    Lógico esvacío()
    Lógico eslleno()
    Entero retorna_dato(i)
    Entero retorna_posición(d)
    Entero donde_insertar(d)
    Entero número_de_elementos()
    ~Vector()        // destructor
Fin(clase Vector)
```

En la definición anterior es bueno hacer notar que hay cuatro métodos, que están precedidos por la palabra *Entero*: retorna_dato(i), retorna_posición(d), número_de_elementos() y donde_insertar(d), y dos métodos que están precedidos por la palabra *Lógico*: esvacío() y eslleno(), lo cual significa que estos métodos retornan datos de tipo entero y lógico respectivamente. Los otros métodos no retornan datos, simplemente ejecutan una tarea sobre el objeto que los invoque.

Teniendo definida dicha clase pasemos a elaborar un programa en el cual seamos usuarios de ella.

```
1.    Vector: a(5), b(7)
2.    entero: d, i
3.    read(d)
4.    while d <> 0 do
5.        a.ins_al_final(d)
6.        read(d)
7.    end(while)
8.    a.imprime()
9.    a.ordena_asc()
10.   a.imprime()
11.   read(d)
12.   i = a.donde_insertar(d)
13.   a.ins_en_posicion(i,d)
14.   read(d)
15.   while d <> 0 do
16.       b.ins_al_principio(d)
17.       read(d)
18.   end(while)
19.   b.imprime()
```

En la instrucción 1 definimos dos objetos de la clase vector: **a** y **b**.

En la instrucción 2 definimos dos variables enteras **d** e **i**.

De las instrucciones 3 a 7 construimos el objeto **a** insertando datos al final.

De las instrucciones 8 a 10 manipulamos el objeto **a**, imprimiéndolo, ordenándolo y volviéndolo a imprimir.

En las instrucciones 11 a 13 se lee un dato y lo insertamos en el sitio que le corresponde en el objeto **a**.

En las instrucciones 14 a 18 construimos el objeto **b** insertando datos al principio y luego lo imprimimos.

Es bueno resaltar que la forma como ejecutamos una operación sobre un objeto:

Objeto.método(parámetros)

Como se podrá observar, elaborar programas utilizando clases ya definidas, es más sencillo y comprensible que si tuviéramos que estar desarrollando los algoritmos cada vez que estemos utilizando variables de alguna clase, vectores, por ejemplo.

Nuestro interés es en dos aspectos: definir clases, con sus correspondientes métodos, y elaborar programas siendo usuarios de dichas clases.

Ocupémonos entonces de escribir algunos de los métodos correspondientes a la clase **Vector** que hemos definido.

Cuando vamos a elaborar los métodos correspondientes a una clase debemos especificar a cuál clase pertenece ese método. Para hacer esta especificación precederemos el nombre del método por el nombre de la clase y cuatro puntos.

En general, la forma como encabezaremos cada método es:

Tipo **Clase::**nombre_del_método(parámetros)

Siendo *Tipo* opcional, y se refiere a la clase de dato que retornará el método en cuestión, si es que retorna algún dato.

Métodos de la clase *Vector*

```
Vector::Vector(n)
    V = nuevo entero(n)   // define vector de datos numéricos enteros
    m = 0
```

```
        for i = 1 to n do
                V(i) = 0
        End(for)
fin(Vector)


Vector::ins_al_final(d)
        if eslleno() then
                write('vector lleno')
                exit
        end(if)
        m = m + 1
        V(m) = d
fin(ins_al_final)


Vector::imprime()
        entero: i
        for i = 1 to m do
                write(V(i))
        end(for)
fin(imprime)


Lógico Vector::esvacio()
        retorne(m == 0)
fin(esvacio)


Vector::ordena()
        entero: i, j, k, t
        For i = 1 to n -1 do
                k = i
                for j = i+1 to n do
                        if V(j) < V(i) then
                                k = j
                        End(if)
                end(for)
                t = V(i)
                V(i) = V(k)
                V(k) = t
        end(for)
Fin(ordena)
```

```
Entero Vector::donde_insertar(d)
        entero: i
        i = 1
        while (i <= m) and (V(i) < d) do
                i = i +1
        end(while)
        retorne(i)
fin(donde_insertar)


Vector::ins_en_posición(i,d)
        entero: k
        if es_lleno() then
                write('vector lleno')
                retorne
        end(if)
        k = m
        while k >= i do
                V(k+1) = V(k)
        k = k - 1
        end(while)
        m = m+1
        V(i) = d
fin(ins_en_posicion)


Lógico Vector::eslleno()
        retorne(m == n)
fin(eslleno)


Entero Vector::retorna_dato(i)
        retorne(V(i))
fin(retorna_dato)


Entero Vector::retorna_posición(d)
        entero: i
        i = 1
        while (i <= m) and (V(i) <> d) do
                i = i + 1
        end(while)
        if (i <= m) then
                retorne(i)
```

```
        else
                retorne(0)
        end(if)
fin(retorna_posicion)


Vector::borra(d)
        entero:  i
        i = retorna_posición(d)
        If  i = 0  then
                write('dato no existe')
                retorne
        end(if)
        while  i < m  do
                V(i) = V(i+1)
                i = i+1
        end(while)
        m = m – 1
fin(borra)


Vector::asigna_dato(i,d)
        V(i) = d
fin(retorna_dato)
```

Es bueno resaltar que cuando dentro de un método definido para una clase se invoca otro método correspondiente a la misma clase no es necesario hacer esta invocación indicando sobre cuál objeto se ejecuta el método, ya que él se efectúa sobre el objeto con el cual ha sido invocado dicho método. Es decir, en el método **borra(d)** se invoca el método **retorna_posición(d)**, y este método actuará sobre el objeto que invocó el método **borra(d)**.

Si quisiéramos ejecutar **retorna_posición(d)** sobre otro objeto diferente al que invocó **borra(d)**, allí, sí será necesario preceder la llamada a **retorna_posición(d)** por el objeto sobre el cual se quiere ejecutar **retorna_posición(d)**.

Para aclarar esta situación consideremos que queremos desarrollar un método que construya un nuevo objeto **c** de la clase vector y que este objeto sea el resultado de intercalar los datos correspondientes a los objetos **a** y **b** también de la clase vector.

Dicho método lo definimos así: Vector **Vector::**intercala(b)

Y las instrucciones correspondientes a dicho método son:

```
1. Vector Vector::intercala(b)
2.     Vector: c(20)    // define un nuevo objeto de clase vector: c
3.     Entero: i, j, k
4.     i = 1
5.     j = 1
6.     k = 0
7.       while (i <= número_de_elementos())
                  and (j <= b.número_de_elementos()) do
8.             k = k + 1
9.                   casos de  comparar(retorna_dato(i), b.retorna_dato(j))
10.                        :<:     c.ins_en_posición(k, retorna_dato(i))
11.                                i = i + 1
12.
13.                        :>:     c.ins_en_posición(k, b.retorna_dato(j))
14.                                j = j + 1
15.
16.                        :=:     c.ins_en_posición(k, retorna_dato(i))
17.                                i = i + 1
18.                                j = j + 1
19.                  fin(casos)
20.    end(while)

21.    while  (i <= número_de_elementos())  do
22.           k = k +1
23.           c.ins_en_posición(k, retorna_dato(i))
24.    end(while)

25.    while  (j <= b.número_de_elementos())  do
26.           k = k + 1
27.           c.ins_en_posición(k, b.retorna_dato(j))
28.    end(while)
29.    c.asigna_m(k)
30.    retorne(c)
31. fin(intercala)
```

En el anterior algoritmo utilizamos un subprograma llamado *comparar*, el cual recibe como parámetros dos datos y retorna un dato tipo *carácter*, el cual puede ser '<', '>' o '=', dependiendo de si el primer parámetro es menor, mayor o igual que el segundo parámetro.

Utilizamos además, otro método, este sí de la clase vector, llamado **asigna_m(i)**, el cual actualiza el número de elementos del vector construido.

Cuando invocamos este método: c = **a**.intercala(b)

en el método **intercala** es bueno observar bien las siguientes instrucciones:

- En la instrucción 7 se invoca el método **número_de_elementos()** dos veces: en la primera llamada no lo precedemos de ningún objeto, por consiguiente allí nos estamos refiriendo al objeto **a**, mientras que en la segunda llamada lo precedemos del objeto **b**, lo cual significa que retorna el número de elementos del vector del objeto **b**.
- En las instrucciones 9, 10 y 16 también invocamos el método retorna_dato(i), sin precederlo de ningún objeto, lo cual significa que retorna el dato correspondiente a la posición **i** del vector del objeto **a.**

Resumiendo, cuando dentro de un método de alguna clase invocamos otro método de la misma clase, estaremos trabajando con el objeto con el cual fue invocado el método invocado inicialmente.

Cuando queramos utilizar el destructor utilizaremos el método ~vector(), el cual lo invocaremos con la instrucción **delete(objeto de la clase vector)**.

Por consiguiente si queremos destruir el objeto **a** escribiremos **delete(a)**, y el objeto **a** es destruido.

En algunas situaciones, que veremos más adelante, el destructor requerirá de instrucciones.

15.2. LISTAS LIGADAS

Introducción

Para entrar a hablar del manejo dinámico de memoria consideremos brevemente lo que es el manejo estático de la memoria, es decir, los arreglos.

Tratemos el caso de un vector, según la clase definida anteriormente.

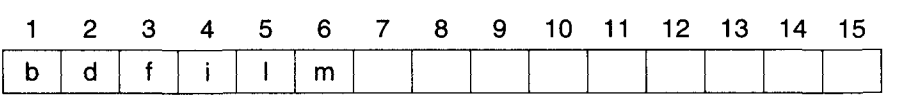


Figura 15.1

En el vector de la figura 15.1 manejamos un conjunto de datos ordenados ascendentemente.

Las operaciones básicas que se realizan sobre ese conjunto de datos son insertar un dato y borrar un dato. Analicemos cada una de ellas.

Si se desea insertar el dato **'g'** en el vector de la figura 1 el proceso a seguir es:
1. Buscar en cuál posición insertarlo. (Método ***donde_insertar(d)***)
2. Insertarlo en la posición correspondiente. (Método ***insertar(i,d)***)

Del primer paso se obtiene que la posición en la cual hay que insertarlo es la posición 4 del vector.

El segundo paso deberá mover los datos desde la posición 4 hasta la posición 6, una posición hacia la derecha y luego asignar a la posición 4 el dato **'g'**.

Si el dato a insertar hubiera sido la letra **'a'** hubiéramos tenido que mover todos los datos del vector.

Si generalizamos, y decimos que el vector tiene **n** datos, en el peor de los casos, hay que mover **n** datos en el vector, o sea, cuando el dato a insertar deba quedar de primero, lo cual implica que el algoritmo de inserción tendrá orden de magnitud **O(n)**.

Si deseamos borrar un dato de un vector los pasos a seguir son:
1. Buscar en cuál posición se halla el dato a borrar. (Método ***retorna_posición(d)***).
2. Borrar el dato del vector. (Método ***borra(d)***)

En caso de que el dato a borrar fuera la letra **'f'**, el primer paso me retorna 3, o sea, la posición en la cual se halla la letra **'f'**.

El segundo paso moverá los datos desde la posición 4 hasta la 6 una posición hacia la izquierda.

Si el dato a borrar hubiera estado en la posición 1 del vector y el vector tiene **n** datos, entonces habrá que mover **n-1** datos hacia la izquierda, y el orden de magnitud de dicho algoritmo es **O(n)**.

De lo expuesto anteriormente, los algoritmos de inserción y borrado tienen orden de magnitud **O(n)**, y esto en el manejo de grandes volúmenes de información, con operaciones de alta frecuencia, como son insertar y borrar se considera ineficiente.

Por lo tanto se ha buscado una forma alterna de representación en la cual las operaciones de inserción y borrado sean eficientes, es decir, tengan orden de magnitud **O(1).**

Consideremos los siguientes dos vectores que llamamos **DATO Y LIGA.**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
DATO	d	i	b		m	f		l						

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LIGA	6	8	1		0	2		5						

Figura 15.2

En el vector **DATO** almacenamos los datos en posiciones aleatorias.

Físicamente los datos se hallan en desorden. Nos interesa tenerlos ordenados lógicamente.

Para ello debemos conocer en cuál posición del vector se halla el primer dato. En nuestro ejemplo, el primer dato se halla en la posición 3 del vector. Utilizaremos una variable, llamémosla **L**, cuyo valor es 3. Es decir, el hecho de que **L** valga 3 significa: el primer dato se halla en la posición 3 del vector **DATO**.

Nos interesa saber en cuál posición se halla el siguiente dato, para ello utilizamos la posición 3 del vector **LIGA**.

El siguiente dato, que es la **'d'**, se halla en la posición 1 del vector **DATO**, por tanto en la posición 3 del vector **LIGA** tendremos un 1.

El hecho de que en la posición 3 del vector **LIGA** haya un 1 significa que el siguiente dato se halla en la posición 1 del vector **DATO.**

Para conocer el siguiente a la **'d'** utilizamos la posición 1 del vector **LIGA**. Allí encontramos un 6, lo que significa que el siguiente dato a la **'d'** se encuentra en la posición 6 del vector **DATO**.

En general, para un dato que se halle en la posición **i** del vector **DATO**, la correspondiente posición **i** del vector **LIGA** indica en cuál posición del vector **DATO** se halla el siguiente dato.

Los textos de computadores acostumbran presentar la situación descrita anteriormente, de la siguiente forma:

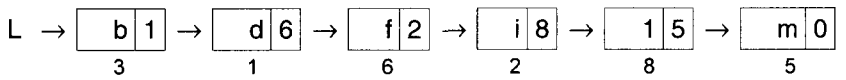


Figura 15.3

Como nuestro objetivo es trabajar programación orientada a objetos, y para ello hacemos uso de clases con sus respectivos métodos, comencemos definiendo la clase en la cual manipulemos el registro que acabamos de definir.

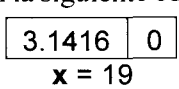
Dicha clase la denominaremos **nodo_simple**

```
Clase nodo_simple
   Privado:
      carácter: dato
      nodo_simple: liga

   Público
      nodo_simple()    // constructor
      retorna_dato()
      retorna_liga()
      asigna_dato(d)
      asigna_liga(x)
      ~nodo_simple()    // destructor
fin(clase nodo_simple)
```

Para hacer uso de esta clase consideremos el siguiente ejemplo, en el cual el dato ha sido definido de tipo real.:

```
d = 3.1416
x = nuevo nodo_simple(d)
```

y obtenemos un registro **x** con la siguiente configuración:



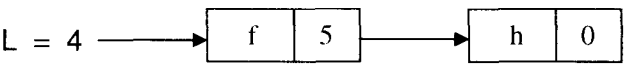
en la cual 19 es la dirección del registro que suministró el sistema operativo. Y si nos queremos referir a sus campos. lo haremos así:

d = x.retorna_dato() y **d** queda valiendo 3.1416
z = x.retorna_liga() y **z** queda valiendo 0

Un ejemplo más completo es:

```
nodo_simple:  x, y, L
read(d)                          // supongamos que leyó la letra 'f'
x = nuevo nodo_simple(d)         // digamos que envió el nodo 4
read(d)                          // supongamos que leyó la letra 'h'
y = nuevo nodo_simple(d)         // digamos que envió el nodo 5
x.asigna_liga(y)
L= x
```

El resultado gráfico al ejecutar estas instrucciones es:



Veamos entonces cómo serían las instrucciones correspondientes a los métodos definidos.

```
Métodos de la clase nodo_simple
nodo_simple::nodo_simple(d)
        dato = d
        liga = 0
fin(nodo_simple)

carácter nodo_simple::retorna_dato()
        retorne(dato)
fin(retorna_dato)

nodo_simple nodo_simple::retorna_liga()
        retorne(liga)
fin(retorna_liga)

nodo_simple::asigna_dato(d)
        dato = d
fin(asigna_dato)

nodo_simple::asigna_liga(x)
        liga = x
fin(asigna_liga)
```

Teniendo definida la clase **nodo_simple,** la cual define el elemento básico para una lista ligada, pasemos a considerar las operaciones que podremos efectuar con una lista ligada.

Operaciones sobre listas ligadas. Las operaciones fundamentales sobre listas ligadas son:

1. Recorrer la lista ligada.
2. Insertar un registro en la lista ligada.
3. Borrar un registro de la lista ligada.

Recorrer la lista ligada. Consideremos la lista de la figura 3, y digamos que sólo nos interesa imprimir los datos que hay en ella.

Para lograr esto requerimos de una variable auxiliar, llamémosla **p**.

Inicialmente, a **p** le asignamos el contenido de **L.**
p queda valiendo 3.

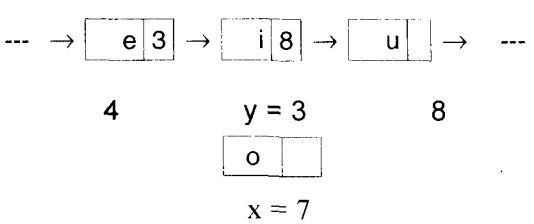
Para imprimir el contenido del campo de **DATO** del registro **p** escribimos **write(p.retorna_dato())** y escribe el dato **'b'**.

Para trasladarnos al siguiente registro escribimos: **p = p.retorna_liga(),** es decir, a la variable **p** se le asigna lo que hay en el campo de liga del registro **p.**

p queda valiendo 1.

Imprimimos el dato del registro **p** y continuamos repitiendo el proceso hasta que **p** sea cero.

Insertar un registro en la lista ligada. Para insertar un registro en una lista ligada se requiere conocer a continuación de cuál registro es que hay que efectuar la inserción. Llamemos **y** este registro, y llamemos **x** el registro a insertar. Consideremos la siguiente figura:



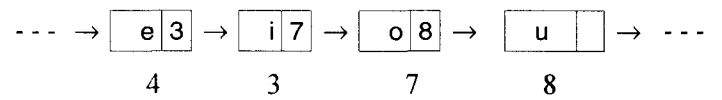
Insertar el registro **x** (**x=7**), a continuación del registro **y** (**y=3**) implica que cuando lleguemos al registro **y** debemos trasladarnos hacia el registro **x**, o sea que el campo de liga del registro **y** debe quedar valiendo 7, y cuando estemos en el

registro **x** debemos trasladarnos hacia el registro 8, es decir, que el campo de liga del registro **x** debe quedar valiendo 8.

Para lograr lo anterior debemos suministrar las siguientes instrucciones:

```
x.asigna_liga(y.retorna_liga())
y.asigna_liga(x)
```

y la lista queda:



Estrictamente hablando, esas son las únicas instrucciones necesarias para insertar un registro **x,** a continuación de un registro **y,** en una lista ligada.

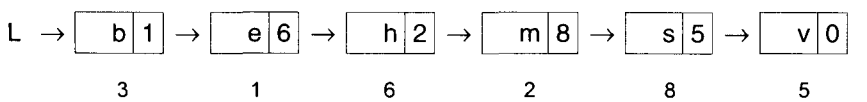
Planteemos ahora el problema más completo:

tenemos una lista ligada, se leyó un dato y hay que insertarlo en la lista.

Los pasos a seguir son:

1. Buscar dónde insertar el dato leído, es decir, determinar **y.**
2. Conseguir un nuevo nodo_simple **x**, guardar el dato leído en **x** y luego insertarlo a continuación del registro **y**.

Ocupémonos del primer paso. Consideremos la siguiente lista:



Sea **d** = **'j'** el dato leído.

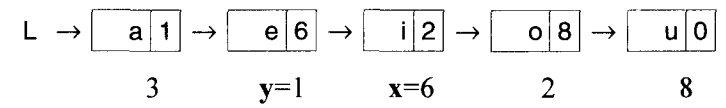
Para buscar dónde insertarlo debemos recorrer la lista comparando el dato de cada registro con el dato leído hasta encontrar un registro cuyo dato sea mayor que **d**, el dato leído.

El nuevo registro debe insertarse antes de ese registro, o sea, a continuación del registro anterior a aquel que tiene un dato mayor. Esto significa que si utilizamos una variable **p** para hacer el recorrido e ir efectuando las comparaciones, debemos utilizar otra variable que permanentemente apunte hacia el registro anterior a **p**. Llamemos esta variable **y**.

Inicialmente a **p** le asignamos el contenido de **L**, y como éste es el primer registro y no tiene anterior, el valor inicial de **y** es cero.

Es bueno hacer notar en este algoritmo que si el dato a insertar debe quedar de primero en la lista, el valor retornado en **y** es cero, y si el dato a insertar debe quedar de último en la lista, entonces **y** quedará apuntando hacia el último registro de la lista ligada.

Borrar un registro de una lista ligada. El concepto fundamental de borrar un registro de una lista ligada es: borrar un registro **x** de una lista ligada significa desconectarlo de la lista. Si tenemos la siguiente lista:



y queremos borrar el registro **x**, significa que cuando estemos ubicados en el registro 1 (**y**=1), debemos pasarnos hacia el registro 2, en vez de hacia el registro 6. Esto implica que debemos conocer cuál es el registro anterior a **x**.

Llamemos entonces **y**, el registro anterior a **x**. La instrucción para desconectar el registro **x** de la lista ligada es simplemente:

```
y.asigna_liga(x.retorna_liga())
```

Estrictamente hablando, esa es la única instrucción necesaria para borrar un registro **x** de una lista ligada.

Consideremos el proceso completo.

Sea **d** la variable que contiene un dato leído que se desea borrar de una lista ligada **L**. Los pasos a seguir son:

1. Determinar **x** e **y.**
2. Borrar **x**.

Para determinar **x** e **y** basta recorrer la lista comparando el dato del registro **x** con el dato leído **d** hasta que sean iguales. Cuando esto suceda hemos determinado los valores de **x** e **y**.

Inicialmente el valor que se le asigne a **x** es **L** y el valor de **y** es cero.

Considerando los anteriores procesos pasemos a definir una clase lista simplemente ligada, la cual llamaremos **LSL.**

```
Clase LSL
   Privado:
      Nodo_simple: L

   Público:
      LSL()                          // constructor
      recorre()
      nodo_simple  donde_insertar(d)
      insertar(d)
      insertar(d, y)
      nodo_simple  buscar_nodo(d)
      borrar(d)
      nodo_simple  primero()
      nodo_simple  último()
      nodo_simple  anterior(x)
      lógico fin_de_lista(x)
      lógico es_vacía()
      ~LSL()                         // destructor
fin(clase LSL)
```

Como se podrá observar, hemos definido dos métodos con el nombre insertar. La diferencia entre estos dos métodos consiste en que uno sólo tiene un parámetro, mientras que el otro tiene dos parámetros. En el primer método, *insertar(d)*, el algoritmo correspondiente a este método buscará el sitio en el cual hay que efectuar la inserción, de acuerdo a algún criterio de ordenamiento, mientras que en el segundo método, *insertar(d, y)*, habrá que insertar un nodo con dato **d** a continuación del nodo **y**, pasado como parámetro. Esta cualidad de poder definir métodos diferentes con el mismo nombre se denomina *polimorfismo.*

Veamos ahora los algoritmos correspondientes a dichos métodos.

```
LSL::LSL()
      L = 0
fin(LSL)


LSL::recorre()
      nodo_simple:  p
      p = primero()
```

```
        while  no fin_de_lista(p)   do
                write(p.retorna_dato())
                p = p.retorna_liga()
        end(while)
fin(recorre)


nodo_simple  LSL::donde_insertar(d)
        nodo_simple:  p, y
        p = primero()
        y = anterior(p)
        while  (no fin_de_lista(p))  and  (p.retorna_dato() < d)   do
                y = p
                p = p.retorna_liga()
        end(while)
        retorne(y)
fin(donde_insertar)


LSL::insertar(d)
        nodo_simple:  x
        x = nuevo nodo_simple(d)
        y = donde_insertar(d)
        if  y  <>  0  then
                x.asigna_liga(y.retorna_liga())
                y.asigna_liga(x)
        else
                x.asigna_liga(L)
                L = x
        end(if)
fin(insertar)



LSL::insertar(d, y)
        nodo_simple: x
        x = nuevo nodo_simple(d)
        if  y = 0  then
                x.asigna_liga(L)
                L = x
        else
                x.asigna_liga(y.retorna_liga())
                y.asigna_liga(x)
```

```
        end(if)
fin(insertar)


nodo_simple  LSL::buscar_nodo(d)
        nodo_simple:  x
        x = primero()
        while  (no fin_de_lista(x))  and  (x.retorna_dato()) <> d  do
                x = x.retorna_liga()
        end(while)
        if  fin_de_lista(x)  then
                retorne(0)
        else
                retorne(x)
        end(if)
fin(buscar_nodo)


LSL::borrar(d)
        nodo_simple:  x, y
        x = buscar_nodo(d)
        if  x = 0  then
                write('dato no existe')
                retorne()
        end(if)
        if  x <>  L  then
                y = anterior(x)
                y.asigna_liga(x.retorna_liga())
        else
                L = L.retorna_liga()
        end(if)
        delete(x)
fin(borrar)


nodo_simple LSL::primero()
        retorne(L)
fin(primero)


nodo_simple LSL::último()
        nodo_simple: x
        if  esvacia()  then
```

```
            retorne(0)
        end(if)
        x = L
        while x.retorna_liga() <> 0 do
            x = x.retorna_liga()
        end(while)
        retorne(x)
fin(último)


nodo_simple LSL::anterior(x)
        nodo_simple: y
        if (x = L) or (L = 0) then
            retorne(0)
        end(if)
        y = primero()
        while y.retorna_liga() <> x do
            y = y.retorna_liga()
        end(while)
        retorne(y)
fin(anterior)


lógico LSL::fin_de_lista(x)
        retorne(x = 0)
fin(fin_de_lista)


lógico LSL::esvacia()
        retorne(L = 0)
fin(esvacia)


LSL::~LSL()
        nodo_simple: x
        while no fin_de_lista(L) do
            x = L
            L = L.retorna_liga()
            delete(x)
        end(while)
fin(~LSL)
```

Como se podrá observar los métodos para insertar y borrar son algoritmos con orden de magnitud **O(1)**, lo cual mejora sustancialmente estas operaciones, con respecto a la representación de datos en un vector.

Construcción de listas ligadas. Pasemos ahora a considerar cómo construir listas ligadas.

Básicamente hay tres formas de construir una lista ligada. Una en que los datos queden ordenados a medida que la lista se va construyendo, otra es insertando registros siempre al final de la lista y otra es insertando los registros siempre al principio de la lista.

Para la primera forma de construcción, que los datos queden ordenados a medida que se va construyendo la lista, un algoritmo general es:

```
LSL: a
a = nueva LSL()
mientras haya datos por leer haga
        lea(d)
a.insertar(d)
fin(mientras)
```

Es decir, basta con plantear un ciclo para lectura de datos e invocar el método *insertar(d)* desarrollado previamente.

Para la segunda forma de construcción, insertando registros siempre al final de la lista, un algoritmo es:

```
LSL: a
nodo_simple: y
a = nueva LSL()
mientras haya datos por leer haga
        lea(d)
        y = a.último()
        a.insertar(d, y)
fin(mientras)
```

Si consideramos que la lista tiene **n** registros, el orden de magnitud de dicho algoritmo es **O(n)**, debido a que por cada registro que haya que insertar hay que recorrer toda la lista para determinar cuál es el último registro. Lo anterior, además de impráctico es ineficiente, ya que sabemos que todo registro debe insertarse siempre al final de la lista.

Para obviar este problema, podemos definir nuestra clase lista simplemente ligada (LSL) con dos variables privadas: **L** que apunta hacia el primer registro de la lista ligada y **ULTIMO**, la cual siempre apuntará hacia el último registro de la lista ligada. (ver ejercicios propuestos)

Para la tercera forma de construcción, insertando registros siempre al principio de la lista, nuestro algoritmo es:

```
LSL: a
a = nueva LSL()
mientras haya datos por leer haga
        lea(d)
        a.insertar(d, 0)
fin(mientras)
```

Hasta aquí hemos desarrollado lo que es básicamente la herramienta lista ligada, con sus operaciones fundamentales: construir la lista, recorrer la lista, insertar un registro y borrar un registro.

Interacción con el sistema operativo. Veamos ahora un poquito referente a la instrucción nuevo, de cuando escribimos x = **nuevo** nodo_simple(d) y al método ***delete*** de la clase **nodo_simple.**

Parte de las funciones del sistema operativo de un computador es administrar la memoria de éste. Dicha administración incluye conocer permanentemente cuáles registros de memoria están libres y cuáles están ocupados, para que así, cuando algún programa que trabaje memoria en forma dinámica, invoque alguno de estos procedimientos su interacción con el sistema operativo sea correcta y eficiente.

Dicha interacción consiste en que el sistema operativo le asignará memoria al programa cuando éste lo solicite y recibirá memoria cuando el programa la libere.

X = nuevo nodo_simple(d): El sistema operativo asigna memoria al programa que hizo la solicitud. La dirección del registro o bloque asignado, retorna en la variable **X**. Para poder efectuar una asignación correcta, el sistema operativo debe conocer cuáles registros de memoria están libres y asignar uno de ellos.

delete(X): Da la orden al sistema operativo de que disponga del registro o bloque **X** y lo marque como libre.

Una forma de conocer cuáles registros están libres y cuáles están ocupados es que el sistema operativo maneja una lista ligada con los registros disponibles.

Si un programa invoca **x = nuevo nodo_simple(d)** entonces el sistema operativo asignará el primer registro de la lista ligada en la que maneja los registros libres, al programa que hizo la solicitud. Esta lista ligada consta de todos los registros disponibles, y una vez asignado al programa, lo borrará de la lista de disponibles, es decir, lo desconecta de dicha lista.

Si se invoca **delete(x)**, el sistema operativo inserta un registro al principio de la lista de registros disponibles.

Las operaciones de inserción y borrado las hace en un solo extremo, lo que significa que la lista de disponibles la trabaja como una pila. Los algoritmos para conseguir un registro y liberar un registro se presentan a continuación.

```
sub_programa (x = nuevo nodo_simple(d))
        if disp = 0 then
                write(«no hay registros disponibles»)
                stop
        end(if)
        x = disp
        disp = disp.retorna_liga()
fin(sub_programa).
```

```
sub_programa delete(x)
        x.asigna_liga(disp)
        disp = x
fin(sub_programa).
```

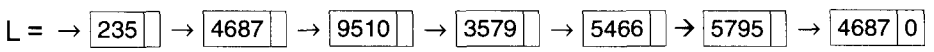
Un ejemplo de aplicación: Manejo de números enteros de alta precisión

Es a veces necesario dentro de algunas aplicaciones manejar números enteros cuya capacidad supera la que manejan los computadores convencionales. Por ejemplo, si se desea conocer exactamente la distancia al sol en milímetros, ninguna máquina de las actuales soportaría esa cantidad de dígitos, por tanto debemos buscar una forma de representación en la cual podamos obtener dicha cifra y además hacer operaciones aritméticas con dichos números.

Primero que todo definamos la forma como representaremos números enteros de esta magnitud utilizando listas ligadas. Si deseamos representar el siguiente número:

235468795103579546657954687

lo almacenaremos en una lista ligada, de a cuatro dígitos por registro de la siguiente forma:



Teniendo definida la representación, el siguiente paso será desarrollar los programas tendientes a manipular los números enteros de alta precisión bajo dicha representación.

Como hemos decidido representar nuestros números de alta precisión como lista ligada, definiremos una clase llamada **alta_precisión**, la cual será derivada de nuestra clase lista simplemente ligada **(LSL)**. Esta característica de poder definir clases derivadas de otras se denomina **herencia**.

En nuestro ejemplo, la clase **alta_precisión** será derivada de la clase **LSL**, lo cual implica que todo objeto definido de **alta_precisión** podrá hacer uso de todos los métodos definidos para la clase **LSL**. Si queremos que los objetos de la clase **alta_precisión** puedan acceder también los datos privados de la clase LSL, debemos definir dichos datos como **protegidos**, en vez de privados.

Al definir la clase alta_precisión como derivada de la clase LSL, se dice que la clase **LSL es la clase base y alta_precisión la clase heredada**.

Los métodos a desarrollar serán:

- **Leer número,**
- **Imprimir número.**
- **Sumar dos números.**
- **Restar dos números.**
- **Multiplicar dos números.**
- **Dividir dos números.**

Definamos entonces nuestra clase **alta_precisión**

```
Clase alta_precisión  derivada de LSL
Público:
        lee_número()
        imprime_número()
        alta_precisión sume(b)
        alta_precisión reste(b)
        alta_precisión multiplique(b)
        alta_precisión divida(b)
```

```
        evalue(d1, d2, acarreo)
        reversa_lista()
        sumele(b)
fin(alta_precisión)
```

El método **lee_número()** aceptará como entrada una hilera de caracteres, la procesará y construirá una lista simplemente ligada, almacenando de a cuatro dígitos por registro. En el capítulo correspondiente a manejo de caracteres presentaremos este algoritmo.

El método **imprime_número()** imprimirá el número representado en una lista.

Los métodos para sumar, resta, multiplicar y dividir, crearán una nueva lista en la cual se tendrá el resultado correspondiente a efectuar cada operación con el objeto de llamada y el objeto **b** entrado como parámetro.

Teniendo definida esta clase podremos escribir un programa para manejar números de alta precisión. Un programa podría ser:

```
alta_precisión:  a, b, c
a = nuevo alta_precisión
b = nuevo alta_precisión
a.lee_número()
b.lee_número()
c = a.sume(b)
a.imprime_número()
b.imprime_número()
c.imprime_número()
Ocupémonos ahora del algoritmo de sumar.
```

Si deseamos sumar 4358795 con 62759, en nuestra aritmética, la suma se efectúa así:

```
  4358795
+   62759
  4421554
```

es decir, comenzamos sumando primero los dígitos menos significativos y nos movemos en ambos números de derecha a izquierda teniendo en cuenta si se lleva algún acarreo o no.

Lo anterior implica que para poder efectuar la suma en nuestra representación debemos ubicarnos en el último registro de cada lista y luego devolvernos en ellas

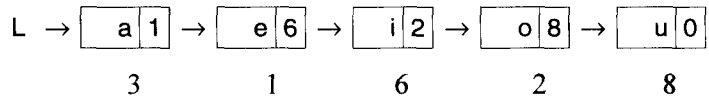
simultáneamente efectuando la suma del número contenido en cada registro, teniendo en cuenta el acarreo. Pero, como ya hemos visto anteriormente, las lista ligadas que tenemos definidas sólo se pueden recorrer en un solo sentido: de izquierda a derecha.

Por consiguiente debemos definir un mecanismo que nos permita recorrer las listas ligadas en reversa, es decir, de derecha a izquierda.

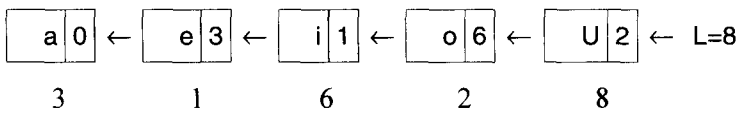
Lo que haremos será reversar los apuntadores de la lista, es decir el campo de liga de cada registro, apuntará hacia el registro anterior y no hacia el siguiente.

Llamemos este método **reversa_lista()**. El efecto de este algoritmo será:

Si tenemos la siguiente lista



y le aplicamos el algoritmo **reversa_lista()**, la lista quedará:

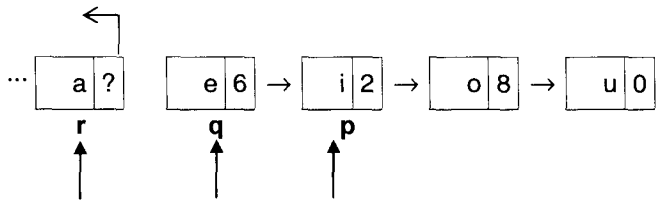


y la podremos recorrer de derecha a izquierda.

Ocupémonos de elaborar el algoritmo correspondiente a dicho método

Para lograr este objetivo sólo necesitamos conocer la dirección de tres registros consecutivos: llamémoslos **r, q,** y **p,** donde **r** apunta hacia el registro anterior a **q**, **q** apunta hacia el registro anterior a **p**, y **p** es la variable con la cual iremos recorriendo la lista.

esquemáticamente tendríamos la siguiente situación:



y el proceso de reversar apuntadores se efectuaría con las siguientes instrucciones:

```
q.asigna_liga(r)
r = q
q = p
p = p.retorna_liga()
```

En las cuales, la primera instrucción reversa el apuntador de **q** y en las tres siguientes se avanza con **r**, **q** y **p**.

Entendiendo las anteriores instrucciones, el siguiente paso es incrustarlas dentro de un ciclo de recorrido de la lista y habremos reversado los apuntadores de la lista.

Nuestro algoritmo será:

```
alta_precisión::reversa_lista()
        nodo_simple: p,q,r
        p = primero()
        q = 0
        while  (no fin_de_lista(p))  do
                r = q
                q = p
                p = p.retorna_liga()
                q.asigna_liga(r)
        end(while)
        L = q
fin(subprograma)
```

Veamos cómo sería el algoritmo para nuestro método de suma:

```
alta_precision::sume(b)
      nodo_simple:  p, q
       entero:  acarreo
       reversa_lista()
       b.reversa_lista()
       c = nuevo alta_precisión()
       p = primero()
       q = b.primero()
       acarreo = 0
       while  (no fin_de_lista(p)) and (no b.fin_de_lista(q))  do
            c.evalue(p.retorna_dato(), q.retorna_dato(), acarreo)
```

```
            p = p.retorna_liga()
            q = q.retorna_liga()
        end(while)
        while (no fin_de_lista(p))
            c.evalue(p.retorna_dato(), 0, acarreo)
            p = p.retorna_liga()
        end(while)
        while (no fin_de_lista(p))
            c.evalue(0, q.retorna_dato(), acarreo)
            q = q.retorna_liga()
        end(while)
        if (acarreo <> 0)
            c.insertar(acarreo, 0)
        end(if)
        reversa_lista()
        b.reversa_lista()
        retorne(c)
fin(sume)
```

Nuestro algoritmo de suma utiliza un método denominado **evalúe**, el cual recibe como parámetros los datos a sumar, efectúa la suma de ellos, separa los últimos cuatro dígitos e inserta un registro al principio de la lista correspondiente al objeto que está creando, es decir, el objeto **c**.

```
alta_precision::evalue(d1, d2, acarreo)
    entero: s
    s = d1 + d2 + acarreo
    acarreo = s / 10000
    s = s % 10000
    insertar(s, 0)
fin(evalue)
```

Es conveniente resaltar que el tercer parámetro, acarreo, debe ser un parámetro por referencia.

El operador % retorna el residuo de una división entera.

Presentamos a continuación el algoritmo correspondiente al método de imprime_número().

```
alta_precisión::imprime_numero()
    nodo_simple: p
    p = primero()
```

```
      write(p.retorna_dato())
      p = p.retorna_liga()
      while  no fin_de_lista(p)  do
        if   p.retorna_dato() < 10  then
             write('000')
        else
             if   p.retorna_dato() < 100   then
                  write('00')
             else
                  if   p.retorna_dato() < 1000   then
                       write('0')
                  end(if)
             end(if)
        end(if)
      write(p.retorna_dato()
      p = p.retorna_liga()
fin(imprime_número)
```

Teniendo ya estos sub_programas elaborados podemos escribir otro algoritmo así:

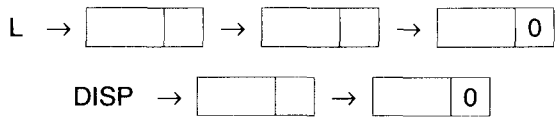
```
alta_precisión:  a, b, c
a.lee_número()
b.lee_número()
c.lee_número()
d = a.sume(b)
e = d.sume(c)
a.imprime_número()
b.imprime_número()
c.imprime_número()
e.imprime_número()
delete(d)
```

En el anterior subprograma nuestro interés ha sido sumar tres números, y como sólo disponemos de un algoritmo que suma dos números utilizamos una lista auxiliar, la cual hemos llamado **d**, y el resultado lo obtenemos en la lista que llamamos **e**.

La última instrucción de ese algoritmo ha sido **delete(d)**, la cual invoca el destructor de la clase LSL (lista simplemente ligada).

Ocupémonos entonces de este algoritmo cuya función es retornar todos los registros de una lista ligada a la lista de disponibles.

Liberación de listas ligadas. Sea **L** el apuntador hacia el primer registro de la lista que se desea liberar; sea **DISP** el apuntador hacia el primer registro de la lista ligada de registros disponibles.



El siguiente algoritmo retorna todos los registros de la lista **L** a la lista de disponibles, uno por uno.

```
LSL::~LSL()
      nodo_simple: x
      while no fin_de_lista(L) do
            x = L
            L = L.retorna_liga()
            delete(x)
      end(while)
fin(~LSL)
```

Así, que si la lista tiene **n** registros el algoritmo será de orden de magnitud O(n). Tener un algoritmo que devuelva, uno por uno, los registros de una lista ligada es ineficiente. Sería mucho más eficiente devolver todos los registros de una sola vez, aprovechando que los registros de una lista ligada están conectados.

Si tenemos la lista **L** y quiero llevar sus registros a la lista de disponibles (**DISP**), sin tener que hacerlo uno por uno, debe recorrerse la lista **L** para encontrar su último registro y ponerlo a apuntar hacia el primer registro de la lista **DISP**. Luego la nueva lista **DISP** será **L** y tendrá la misma dirección de memoria que **L**.

Veamos el algoritmo:

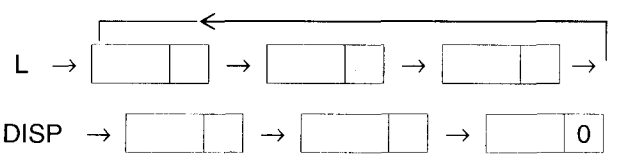
```
LSL::~LSL()
      nodo_simple: x
      x = ultimo()
      x.asigna_liga(disp)
      disp = L
      L = 0
fin(~LSL)
```

Este algoritmo conecta todos los registros de la lista **L** a la lista de disponibles. El orden de magnitud es **O(n)**, ya que la ejecución del método **último()** implica recorrer la lista en su totalidad, sin embargo, su contador de frecuencias es menor que el algoritmo dado inicialmente.

Sin embargo, para procesos que se presentan con mucha frecuencia, lo ideal es tener un algoritmo con orden de magnitud **O(1)**. Para obtener esto, basta con hacer una pequeña modificación a la representación de la lista ligada: hacerla circular, o sea que el campo de liga del último registro ya no será cero sino que apuntará hacia el primer registro de la lista ligada.

El procedimiento de devolver una lista en representación de lista ligada circular tendrá orden de magnitud **O(1)**.

El proceso consiste en encadenar **DISP** (Lista de disponibles) al primer registro de la lista **L** y asignarle a **DISP** el valor del segundo registro de la lista **L**. Para ello necesitamos guardar la dirección del segundo registro de la lista **L** y lo haremos en una variable que llamaremos **x**.




```
LSLC::devolver_lista_circular()
        x = L.retorna_liga()
        L.asigna_liga(disp)
        disp = x
        L = 0
        x = 0
fin(sub_programa)
```

Para lograr dicho objetivo se guarda en una variable **x** el segundo registro de la lista **L**. La variable **DISP** apuntará hacia este registro y el campo liga del registro **L** apuntará hacia el registro que inicialmente era **DISP.**

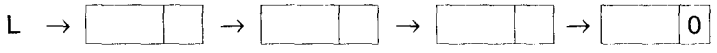
Por consiguiente, debido a que el procedimiento de retornar lista es bastante más eficiente con listas circulares tomaremos la decisión de representar las listas ligadas como circulares cuando la operación de devolver lista tenga alta frecuencia.

Esta decisión la podremos aplicar entonces a la representación de números de alta precisión, ya que en un paquete de este estilo, liberar listas será un proceso con alta frecuencia de ejecución.

Sin embargo, esta decisión afectará todos los otros algoritmos del manejo de números de alta precisión y tocará evaluar si se justifica representar los números de alta precisión en listas circulares.

Cuando se recorre una lista ligada hay dos situaciones fundamentales que se deben controlar: una la situación de lista vacía y la otra la situación de terminación de recorrido de la lista.

Diferentes tipos de listas ligadas y sus características
Listas simplemente ligadas:



Es la clase que hemos venido trabajando hasta ahora. Analicemos más en detalle el método que tenemos para recorrer e imprimir objetos de esta clase.

```
LSL::recorre()
      nodo_simple:  p
      p = L
      while  (no fin_de_lista(p))  do
            write(p.retorna_dato())
            p = p.retorna_liga()
      end(while)
fin(recorre)
```

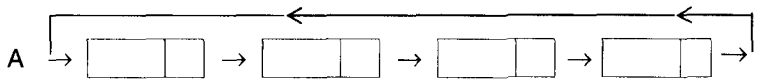
El algoritmo recorre e imprime el contenido de cada registro de la lista. Aquí se controla la situación de lista vacía, **L = 0**, con la misma instrucción con que se controla la terminación de recorrido de la lista: **while (no fin_de_lista(p)).**

Es decir, con una sola instrucción estamos controlando dos situaciones.

Resumiendo, cuando tenemos una lista simplemente ligada, las condiciones son:

lista vacía: L = 0
terminación de recorrido: p = 0

Lista Simplemente Ligada Circular



Para recorrer la lista no podemos elaborar un algoritmo como en el caso de la lista simplemente ligada porque cuando tenemos la lista circular ningún registro tendrá campo de liga cero (0), por tanto la situación de terminación de recorrido no se podrá controlar con la instrucción **while p <> 0**.

Cuando tenemos la lista circular el recorrido se termina cuando la variable auxiliar para el recorrido, es decir **p**, vuelva a ser igual a **L**. Plantear el ciclo con la instrucción **while p ≠ L** no sería correcto porque la primera instrucción es asignarle a la variable **p** el contenido de la variable **L**, y entonces preguntar si **p** es diferente de **L** daría como resultado falso y nunca entraría al ciclo.

Por consiguiente debemos plantear un ciclo que permita entrar primero al ciclo y después preguntar por la condición de si **p** es diferente de **L**. Para ello disponemos del ciclo **Do ... While** (condición) . Nuestro algoritmo sería:

```
LSLC::recorrer_lista_circular()
nodo_simple:  p
      p = L
      do
          write(p.retorna_dato())
          p = p.retorna_liga()
      while  p <> L
fin(recorre_lista_circular)
(LSLC  para clase  Lista Simplemente Ligada Circular)
```

Lamentablemente este algoritmo no controla la situación de lista vacía.

En caso de que la **L** del objeto **LSLC** entre valiendo 0, la primer vez, **p** queda valiendo 0, entra al ciclo y cuando vaya a ejecutar write(p.retorna_dato()) nuestro programa cancela.

Para que el programa NO cancele debemos añadir instrucciones, como preguntar si **p <> 0**, antes del ciclo **DO**, lo cual, implica que en todas las situaciones donde haya que recorrer una lista habrá que controlar dicha situación con más instrucciones, lo cual va en detrimento del programa.

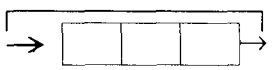
¿Qué pasó entonces?: que por querer hacer eficiente el método correspondiente al destructor, estamos desmejorando todos los algoritmos en los cuales haya que recorrer alguna lista, porque necesitamos instrucciones diferentes para controlar las situaciones de lista vacía y de terminación de recorrido de la lista.

Nuestro objetivo es poder tener una lista en la cual devolver lista sea eficiente y además controlar las situaciones de lista vacía y de terminación de recorrido con una sola instrucción.

Dicho objetivo lo logramos añadiendo un registro al principio de la lista, el cual llamaremos registro cabeza.

Lista simplemente ligada circular con registro cabeza

El registro cabeza es un registro que, por lo general, no tendrá información correspondiente al objeto que se esté representando en la lista ligada. Siempre será el primer registro de la lista ligada y facilita todas las operaciones sobre la lista: construir la lista, recorrer la lista, insertar un registro, borrar un registro y devolver la lista.

La situación de lista vacía se representa así: L 

El campo de liga del registro cabeza apunta hacia sí mismo.

Realmente, si un registro adicional presenta tantas ventajas, no debemos preocuparnos mucho por el hecho de consumir una posición más de memoria.

Vamos entonces a definir una clase **L**ista **S**implemente **L**igada **C**ircular **C**on **R**egistro **C**abeza (**LSLCCRC**), la cual será derivada de la clase **L**ista **S**implemente **L**igada (**LSL**), por consiguiente podremos hacer uso de los métodos desarrollados en la clase **LSL**.

```
Clase LSLCCRC derivada de LSL
Público:
        LSLCCRC()
        virtual nodo-_simple  ultimo()
```

```
        virtual nodo_simple  primero()
        virtual lógico   fin_de_lista(x)
        virtual lógico   esvacia()
        virtual nodo_simple   anterior(x)
        virtual borrar(d)
        virtual ~lslccrc()
fin(clase LSLCCRC)
```

Observe que estos métodos han sido precedidos de la palabra **virtual**, lo cual significa que cuando se esté ejecutando un programa con objetos **LSL** y/o objetos **LSLCCRC** el computador distinguirá cuál utilizar dependiendo del objeto con el cual se esté trabajando.

```
lslccrc::lslccrc()
        L = nuevo nodo(0)
        L.asigna_liga(L)
fin(LSLCCRC)

nodo_simple lslccrc::primero()
        retorne(L.retorna_liga())
 fin(primero)

lógico  lslccrc::fin_de_lista(x)
        retorne (x = L)
fin(fin_de_lista)

lógico lslccrc::esvacia()
        retorne (L = L.retorna_liga())
fin(esvacía)

nodo_simple lslccrc::ultimo()
        nodo_simple:  p
        if (esvacia())
                retorne(L)

        end(if)
        p = primero()
        while (no fin_de_lista())
                p = p.retorna_liga()
         end(while)
         retorne(p)
fin(ultimo)
```

```
nodo_simple  lslccrc::anterior(x)
        nodo_simple:  q, ant
        ant = l
        q = primero();
        while (no fin_de_lista(q)) and (q <> p)
              ant = q
              q = q.retorna_liga()
       end(while)
        retorne(ant)
fin(anterior)

lslccrc::borrar(d)
       nodo:   x, y
       if (esvacia())
              write('la lista esta vacía')
              retorne
       end(if)
       x = buscar_nodo(d)
       if  x = 0  then
              write('el dato no existe')
              retorne
       end(if)
       y = anterior(x)
       y.asigna_liga(x.retorna_liga())
       delete(x)
fin(borrar)

lslccrc::~lslccrc()
       nodo  p, x
       p = primero()
       while (no fin_de_lista(p))
              x = p;
              p = p->retorna_liga()
              delete(x)
       end(while)
       delete(L)
       L = 0
fin(destructor)
```

Para recorrer la lista se comienza en el segundo registro, es decir, a **p** se le asigna **liga(L)**. La lista se termina de recorrer cuando **p** sea igual a **L**.

Por consiguiente, si la lista está vacía, **p** queda valiendo **L** y no entra al ciclo de recorrido.

Con una sola condición estamos controlando dos situaciones.
También puedo tener lista simplemente ligada no circular con registro cabeza.

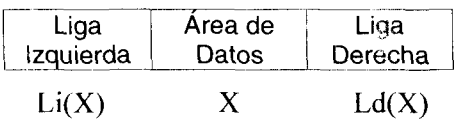
El caso es, que de acuerdo al problema en particular que se quiera desarrollar, utilizando listas ligadas, se tomará la decisión de cuál tipo de representación utilizar.

Listas doblemente ligadas

Hasta aquí hemos tratado con listas ligadas que sólo se pueden recorrer en un solo sentido: de izquierda a derecha.

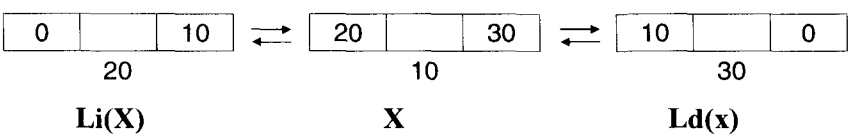
Existen cierto tipo de problemas que exigen que las listas ligadas se puedan recorrer en ambas direcciones: de izquierda a derecha y de derecha a izquierda.

Para lograr esto debemos diseñar registros (nodos) con dos campos de liga: uno que llamaremos **Li** y otro que llamaremos **Ld**. Un esquema general de dicho nodo es:



Li(X): apunta hacia el registro anterior a **x**.
Ld(X): apunta hacia el registro siguiente de **x**.

Como un ejemplo, consideremos el siguiente segmento de lista doblemente encadenada:



El registro anterior a **x** es el registro **Li(x)** y el registro siguiente a **x** es el registro **Ld(x).**

La propiedad fundamental de las listas Doblemente Ligadas es:

Ld(Li(x)) = x = Li(Ld(x))

Lo cual, dicho en palabras es: el campo liga derecha del registro liga izquierda de **x** es el mismo registro **x**, igual que el campo liga izquierda del registro liga derecha de **x**.

Con base en lo anterior vamos entonces a definir la clase nodo_doble:

```
Clase nodo_doble
   Privado:
      carácter: dato
      nodo_doble: li, ld
   Público
      nodo_doble()    // constructor
      retorna_dato()
      asigna_dato(d)
      retorna_li()
      asigna_li(x)
      retorna_ld()
      asigna_ld(x)
      ~nodo_doble()    // destructor
fin(clase nodo_doble)
```

Los algoritmos correspondientes a cada uno de estos métodos se dejan como ejercicio al estudiante.

Pasemos entonces a definir la clase **L**ista **D**oblemente **L**igada (**LDL**)

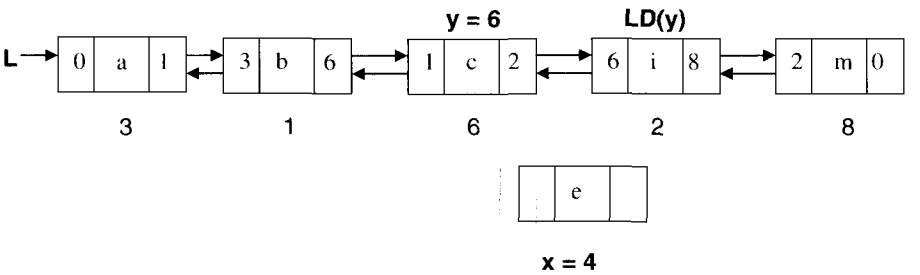
Al manejar listas Doblemente Ligadas, las operaciones básicas son las mismas que en las listas simplemente ligadas: Insertar un registro, borrar un registro y recorrer la lista.

Expliquemos brevemente estos procesos:

Recorridos sobre la lista: los algoritmos son los mismos que para recorrer una lista simplemente ligada teniendo en cuenta qué campo de liga utilizar cuando se vaya a avanzar sobre la lista:

Para movernos de Izquierda a Derecha utilizamos el campo liga derecha de **p**.
Para movernos de Derecha a Izquierda utilizamos el campo liga izquierda de **p**.

Insertar: un registro **x** a continuación de un registro **y**.



En nuestros algoritmos, insertamos un registro **x** a continuación de un registro **y**.

Y: Dirección del registro a continuación del cual se insertará un nuevo registro. El campo de liga izquierda del nodo **x** debe quedar valiendo 6 (**x.asigna_li(y)**).

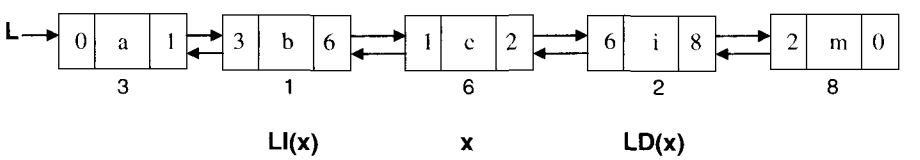
El campo de liga derecha del nodo **x** debe quedar valiendo 2 (**x.asigna_ld(y.retorna_ld()**)

El campo liga izquierda del nodo liga derecha de **y** (nodo 2) debe quedar valiendo 4 (**p =y.retorna_ld()**; **p.asigna_li(x)**). Estas instrucciones conectan el nodo **x** para cuando la lista se esté recorriendo de derecha a izquierda.

El campo liga derecha del nodo **y** debe quedar valiendo 4 (**y.asigna_ld(x)**). Esta instrucción conecta el nodo **x** para cuando se esté haciendo el recorrido de izquierda a derecha.

Borrar un registro de la lista. Sea **x:** Dirección del registro a borrar.

Recuerde que borrar un nodo de una lista ligada consiste en desconectar el nodo de la lista.



Si queremos borrar el nodo **x = 6**, debemos tener en cuenta que la lista es doblemente ligada, por consiguiente el nodo anterior es el nodo **LI(x)** (LI(x)=1) y el nodo siguiente es el nodo **LD(x)** (LD(x)=2).

Para desconectar el registro **x** en sentido de izquierda a derecha lo que hay que hacer es que cuando estemos en el registro **LI(x)** debemos pasarnos hacia el registro **LD(x)**, o sea brincarnos el registro **x**. La instrucción para efectuar esto es:

LD(LI(x))= LD(x)

La cual se lee: al campo liga derecha del registro liga izquierda de **x** le asignamos lo que hay en el campo liga derecha del registro **x**.

Lo cual efectuamos con las instrucciones

```
p = x.retorna_li()
p.asigna_ld(x.retorna_ld())
```

Si estamos recorriendo de derecha a izquierda, cuando estemos ubicados en el registro **LD(x)**, debemos pasarnos hacia el registro **LI(x)**, o sea desconectar el registro **x**. La instrucción para efectuar esto es:

```
LI(LD(x)) = LI(x)
```

Y la escribimos:

```
p = x.retorna_ld()
p.asigna_li(x.retorna_li())
```

En este caso también debemos considerar varias situaciones especiales.

- Que el nodo a borrar sea el primero y que éste sea único.
- Que el nodo a borrar sea el primero y que no sea único.
- Que el nodo a borrar sea el último.
- Que el nodo a borrar esté en un sitio intermedio en la lista.

A continuación presentamos la definición de la clase doblemente ligada.

```
Clase LDL
   Privado:
      Nodo_doble: L

   Público:
      LDL()                              // constructor
      recorre_izq_der()
      recorre_der_izq()
      nodo_doble  donde_insertar(d)
      insertar(d)
      insertar(d, y)
      nodo_doble  buscar_nodo(d)
      borrar(d)
      nodo_doble  primero()
      nodo_doble  último()
      nodo_doble  anterior(x)
      lógico fin_de_lista(x)
      lógico es_vacía()
```

```
      ~LDL()                          // destructor
fin(clase LDL)
```

Los algoritmos para los métodos definidos se presentan a continuación.

Métodos para clase **LDL**.

```
LDL::LDL()
      L = 0
Fin(LDL)


LDL::recorre_izq_der()
      nodo_doble:  p
      p = primero()
      while  (no fin_de_lista(p))  do
            write(p.retorna_dato())
            p = p.retorna_ld()
      end(while)
fin(recorre_izq_der)



LDL::recorre_der_izq()
      nodo_doble:  p
      p = ultimo()
      while  (no fin_de_lista(p))  do
            write(p.retorna_dato())
            p = p.retorna_li()
      end(while)
fin(recorre_der_izq)


nodo_doble  LDL::donde_insertar(d)
      nodo_doble:  p, y
      p = primero()
      y = anterior(p)
      while  (no fin_de_lista(p))  and  (p.retorna_dato() < d)  do
            y = p
            p = p.retorna_ld()
      end(while)
      retorne(y)
fin(donde_insertar)
```

```
LDL::insertar(d, y)
        nodo_doble:  x, p
        x  = nuevo nodo_doble(d)
        casos
                :y = 0:
                        x.asigna_ld(L)
                        if L <> 0 then
                                L.asigna_li(x)
                        end(if)
                        L = x
                :y.retorna_ld() = 0:
                        y.asigna_ld(x)
                        x.asigna_li(y)
                :else:
                        x.asigna_ld(y.retorna_ld())
                        x.asigna_li(y)
                        p = y.retorna_ld()
                        p.asigna_li(x)
                        y.asigna_ld(x)
        fin(casos)
fin(insertar)


LDL::insertar(d)
        nodo_doble:  x, p
        x  = nuevo nodo_doble(d)
        y = donde_insertar(d)
        casos
                :y = 0:
                        x.asigna_ld(L)
                        if L <> 0 then
                                L.asigna_li(x)
                        end(if)
                        L = x
                :y.retorna_ld() = 0:
                        y.asigna_ld(x)
                        x.asigna_li(y)
                :else:
                        x.asigna_ld(y.retorna_ld())
                        x.asigna_li(y)
                        p = y.retorna_ld()
                        p.asigna_li(x)
                        y.asigna_ld(x)
        fin(casos)
fin(insertar)
```

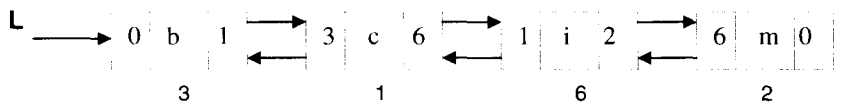
```
LDL::borrar (x)
      if  x = 0 then
            write('dato a borrar no existe')
            retorne
      end(if)
      nodo_doble:  p
      casos
            :x.retorna_li() = 0  and  x.retorna_ld() = 0:  // lista con un registro
                  L = 0

            :x.retorna_li() = 0:                          // borra el primer registro
                  p = x.retorna_ld()
                  p.asigna_li(0)
                  x.asigna_ld(L)

            :x.retorna_ld() = 0:                          // borra el ultimo registro
                  p = x.retorna_li()
                  p.asigna_ld(0)

            :else:
                  p = x.retorna_ld()
                  p.asigna_li(x.retorna_li())
                  p = x.retorna_li()
                  p.asigna_ld(x.retorna_ld())
      fin(casos)
      delete(x)
fin(borrar)


nodo_doble  LDL::buscar_nodo(d)
      nodo_simple:  x
      x = primero()
      while  (no fin_de_lista(x))  and  (x.retorna_dato()) <> d  do
            x = x.retorna_ld()
      end(while)
      if  fin_de_lista(x)  then
            retorne(0)
      else
            retorne(x)
      end(if)
fin(buscar_nodo)
```

```
nodo_doble LDL::primero()
        retorne(L)
fin(primero)


nodo_doble LDL::último()
        nodo_doble: x
        if esvacia() then
                retorne(0)
        end(if)
        x = L
        while x.retorna_ld() <> 0 do
                x = x.retorna_ld()
        end(while)
        retorne(x)
fin(último)


nodo_doble LDL::anterior(x)
        nodo_doble: y
        if (x = L) or (L = 0) then
                retorne(0)
        end(if)
        y = primero()
        while y.retorna_ld() <> x do
                y = y.retorna_ld()
        end(while)
        retorne(y)
fin(anterior)


lógico LDL::fin_de_lista(x)
        retorne(x = 0)
fin(fin_de_lista)


lógico LDL::esvacia()
        retorne(L = 0)
fin(esvacia)


LDL::~LDL()
        nodo_doble: x
        while (no fin_de_lista(L)) do
```

```
            x = L
            L = L.retorna_ld()
            delete(x)
        end(while)
fin(~LSL)
```

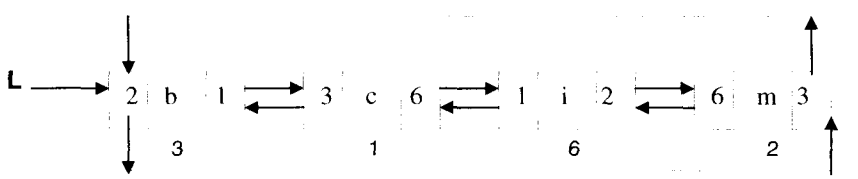
Al igual que las listas simplemente ligadas, con las listas doblemente ligadas se pueden presentar una serie de variaciones:

Listas doblemente ligadas



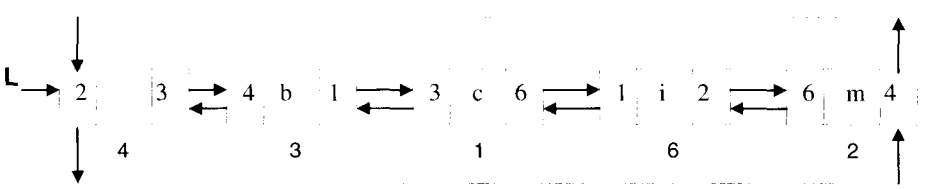
El campo de liga izquierda del primer registro es cero porque el primer registro no tiene anterior, y el campo de liga derecha del último registro también vale cero porque el último registro no tiene siguiente.

Listas doblemente ligadas circulares



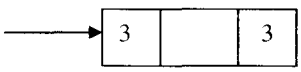
En las listas doblemente ligadas circulares el campo de liga derecha del último registro apunta hacia el primer registro de la lista y el campo de liga izquierda del primer registro apunta hacia el último registro de la lista.

Listas doblemente ligadas circulares con registro cabeza



En las listas doblemente ligadas circulares con registro cabeza se tiene la ventaja de que la representación de la lista vacía tendrá como mínimo el registro cabeza:

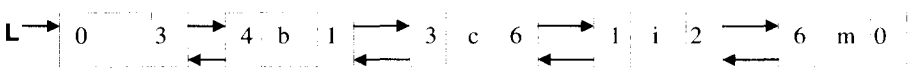
L = 3



LD (L) = 3

Esta es la configuración de la lista vacía cuando se tienen listas doblemente ligadas circulares con registro cabeza.

Listas doblemente ligadas no circulares con registro cabeza



Como en el caso de las listas simplemente ligadas consideraremos la clase **L**ista **D**oblemente **L**igada **C**ircular **C**on **R**egistro **C**abeza (**LDLCCRC**) como derivada de la clase **L**ista **D**oblemente **L**igada (**LDL**).

```
Clase LDLCCRC derivada de LDL
   Público:
      LDLCCRC()                          // constructor
      recorre_izq_der()
      recorre_der_izq()
       nodo_doble  donde_insertar(d)
      insertar(d)
      insertar(d, y)
      nodo_doble  buscar_nodo(d)
      borrar(d)
      nodo_doble  primero()
      nodo_doble  último()
      nodo_doble  anterior(x)
      lógico fin_de_lista(x)
      lógico es_vacía()
      ~LDL()                             // destructor
fin(clase LDL)
```

Los algoritmos correspondientes a los métodos anteriores son:

```
ldlccrc::ldlccrc()
      L = new nodo_doble(0)
      L.asigna_li(L)
      L.asigna_ld(L)
fin(ldlccrc)


nodo_doble  ldlccrc::primero()
      retorne(L.retorna_ld())
fin(primero)

nodo_doble  ldlccrc::ultimo()
       retorne(L.retorne_li()
fin(último)

lógico  ldlccrc::fin_de_lista(p)
     return (p = L)
fin(fin_de_lista)

nodo_doble    ldlccrc::anterior(p)
      nodo_doble:  q,  ant
      ant = L
      q = primero();
      while  (no fin_de_lista(q)  and  (q <> p)  do
          ant = q
          q = q.retorna_ld()
      end(while)
       retorne(ant)
 fin(anterior)


lógico  ldlccrc::esvacia()
       return (L = L.retorna_ld()
fin(esvacía)


ldlccrc::borra(d)
      nodo_doble:  x, y, temp
      if   esvacia()
          write('la lista esta vacia ')
          retorne
      end(if)
      x = primero()
      y = L
```

```
    while (no fin_de_lista(x) and (x.retorna_dato() <> d)
        y = x;
        x = x.retorna_ld()
    end(while)
    if (x = l)
        write('el dato no existe ')
        retorne
    end(if)
    y.asigna_ld(x.retorna_ld())
    x.retorna_ld().asigna_li(y)
    delete(x)
fin(borra)

ldlccrc::~ldlccrc()
    nodo_doble: p, x
    p = primero()
    while (no fin_de_lista(p))
        x = p
        p = p.retorna_ld()
        delete(x)
    end(while)
    delete(L)
    L = 0
fin(~ldlccrc)
```

Retomemos ahora nuestro ejemplo con números de alta precisión.

Si decidimos representar dichos números en listas doblemente ligadas circulares con registro cabeza tendremos varias ventajas:

- Nos evitamos tener que reversar la lista para efectuar las operaciones, puesto que la lista la podemos recorrer en ambos sentidos.
- Con una sola instrucción podemos llegar al último registro de la lista, dado que el campo de liga izquierda del registro cabeza apunta hacia el último nodo de ella.
- El registro cabeza lo utilizamos para poder manejar los números con signo: el campo de dato del nodo cabeza será 0 ó 1: 0 significa que el número es positivo, 1 que el número es negativo, lo cual le da más potencialidad a nuestra clase alta precisión.

Veamos cómo quedaría nuestro algoritmo de suma representando los números enteros de alta precisión en listas doblemente ligadas circulares con registro cabeza.

En este algoritmo la clase alta precisión la hemos definido como derivada de **LDLCCRC**, por consiguiente los métodos utilizados son los correspondientes a esta clase.

```
alta_precision  alta_precision::sume(b)
        nodo_doble:  p, q, r
        entero:  acarreo
        alta_precision:  c
        acarreo = 0
        c = nuevo  alta_precision
        r = c.primero().retorna_li()
        r.asigna_dato(p.retorna_dato())
        p = primero().retorna_li()
        q = b.primero().retorna_li()
        p = ultimo()
        q = b.ultimo()
        while  (no fin_de_lista(p)  and  (no b.fin_de_lista(q)  do
                c.evalue(p.retorna_dato(), q.retorna_dato(), acarreo)
                  p = p.retorna_li()
                q = q.retorna_li()
        end(while)
        while (no fin_de_lista(p)  do
                c.evalue(0, p.retorna_dato(), acarreo)
                p = p.retorna_li()
        end(while)
        while  (no b.fin_de_lista(y))  do
                c.evalue(0, q.retorna_dato(), acarreo)
                q = q.retorna_li()
        end(while)
        if  (acarreo <> 0)  then
                c.insertar(c.primero().retorna_li(), acarreo)
        end(if)
        return(c)
fin(sume)
```

Bibliografía

BECERRA SANTAMARÍA, César. *Estructuras de datos en Pascal*, Computador Limitada, Bogotá, 1989, Capítulo 5.

GOTLIEB AND GOTLIEB. *Data Type and Structures,* Prentice Hall, 1978, New Jersey, Capítulo 3.

HOROWITZ Ellis, SAHNI Sartaj. *Fundamentals of Data Structures*, Pitman Publishing Limited, 1977, London, Capítulos 2, 3, 4, 5 y 6.

Algoritmos, estructuras de datos y programación orientada a objetos

El presente texto es el resultado de un poco más de 15 años impartiendo los cursos de Algoritmos II y Estructuras de datos I en el Departamento de Ingeniería de Sistemas de la Universidad de Antioquia.

El libro tiene un tratamiento eminentemente algorítmico, independiente de los lenguajes de programación.

En los capítulos 1, 2 y 3 se presenta una introducción a los sistemas numéricos, representación de datos en memoria y evaluación de algoritmos, los cuales son temas básicos para comprender mejor el funcionamiento de un computador y la elaboración de algoritmos.

En los capítulos siguientes se presentan las diferentes estructuras utilizadas en el desarrollo de algoritmos: definición de estructuras de datos en abstracto, listas ligadas, matrices dispersas, representación de arreglos en memoria, pilas, colas, manejo de hileras, recursión, árboles y grafos.

Por último se presenta un capítulo de introducción a la programación orientada a objetos, con un tratamiento también independiente de algún lenguaje de programación.

Colección: Textos universitarios
Área: Informática

ECOE
EDICIONES