2cj
24

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

APLICACION DE LAS ESTRUCTURAS DE DATOS EN PASCAL

T E S I S

QUE PARA OBTENER EL TITULO DE:

M A T E M A T I C O

PRESENTA:

GLORIA SANCHEZ LOPEZ

MEXICO, D. F. 1 9 8 7

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

I N D I C E

página

1. INTRODUCCION 1

2. **LA PILA** 2
Implementación de la pila y sus operaciones 5

2.1 Ejemplos de aplicación:
-Validez de la anidación de paréntesis en una expresión aritmética 7
-Manipulación de expresiones aritméticas 11,15
-Factorial de un número n 19
-Permutaciones de n elementos 23

3. **COLAS** 28
Implementación de cola y sus operaciones 31

3.1 Ejemplos de aplicación:
-Simulación 48
-Juegos 85

4. **LISTAS** 33
Implementación de listas y sus operaciones 37,53

4.1 Representación de la pila y cola con listas 46

4.2 Ejemplos de aplicación:
-Simulación 48
-Suma de polinomios 55

4.3 Listas multiencadenadas 58

4.3.1 Ejemplos de aplicación
-Suma de matrices dispersas 59

5. **ARBOLES**

5.1 Arbol binario 64

5.1.2 Ejemplos de aplicación:
-Ordenación de una lista de números 67
-Manipulación de expresiones aritméticas 71

5.1.3 Otra estructura de nodos en árboles binarios y sus operaciones 77

5.2 Arbol binario hilvanado 79
Operaciones básicas 80

5.3 Arbol binario casi completo 82

5.4 Arboles 83
Operaciones básicas 84

página

5.4.1 Ejemplos de aplicacion:
-Juegos 85

6. **GRAFICAS** 91
Operaciones básicas 92

6.1 Ejemplos de aplicación:
-Caminos en una gráfica 93
-Automatas 100

8. **ORDENAMIENTO** 104

8.1 Ordenamiento rapido (Quicksort) 105
Implementacion 106

8.2 Ordenamiento de grupo (Heapsort) 109
Implementación 110

8.3 Ordenamiento de Insercion (Shell) 112
Implementacion 113

8.4 Ordenamiento de concatenación (Merge) 115
Implementacion 116

7. **BUSQUEDA** 119

7.1 Búsqueda secuencial
-Algoritmo 120

7.2 Busqueda secuencial indexada
-Algoritmo 121

7.3 Búsqueda binaria
-Algoritmo 123

7.4 Búsqueda en arbol binario balanceado
-Algoritmo 124

7.5 Funcion de dispersion (hash) 129
-Metodo del cuadrado medio
-Metodo de dobles
-Codigo algebraico

7.5.1 Solucion a las colisiones 130
-Dirección abierta
-Encadenamiento

CONCLUSIONES 131

BIBLIOGRAFIA 132

INTRODUCCION

Con este trabajo se pretende colaborar de manera didáctica en la impartición o aprendizaje del tema de estructura de datos en el área de computación.

En la actualidad el lenguage PASCAL (desarrollado en Zurich, Suiza, por Niklaus Wirth en 1971) ha sido ampliamente aceptado como una excelente herramienta en la enseñanza de la programación debido a que crea buenos hábitos de programación y porque se encuentran disponibles varias implementaciones de ese lenguaje.

PASCAL permite representar de distintas formas las estructuras de datos, lo cual le da al estudiante la oportunidad de apreciar las diferentes alternativas y dificultades que surgen.

El conjunto de programas escrito en lenguaje PASCAL que se presenta, se ejecutó en microcomputadoras compatibles en turbo-pascal.

Las estructuras de datos se utilizan para representar y manipular información relativa a problemas que se resuelven a través de la computadora. Son indispensables en cualquiera de las áreas de la computación, por citar algunas: compiladores, inteligencia artificial, programación de sistemas y bases de datos.

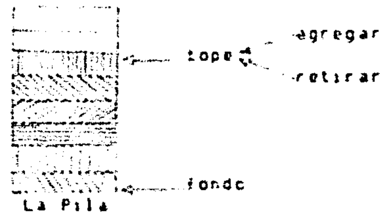
Las estructuras de datos que se verán en el trabajo son: **pila, cola, listas, árboles** y **gráficas**. Asimismo se incluyen los temas de **ordenamiento** y **búsqueda**, los cuales hacen uso de las estructuras de datos y son fundamentales en computación.

En cada una de las estructuras de datos que se presentan aparecen su definición, sus operaciones básicas y sus diferentes formas de representación e implementación, a la vez que se muestra con algunos ejemplos cómo los conceptos abstractos llegan a ser una herramienta muy valiosa. En los resultados de estos ejemplos de aplicación se ilustra cada paso de los algoritmos (esto es la evolución de los programas) para que el lector pueda visualizar y seguir con detalle la manera en que se resuelven los problemas.

LA PILA

DEFINICION

Una pila es una colección ordenada de elementos en la que nuevos elementos pueden ser agregados y desde la cual los elementos pueden ser retirados. Ambas operaciones se realizan por el final, llamado el tope de la pila.



La Pila

CARACTERISTICA PRINCIPAL

El último elemento agregado a la pila es el primero en ser retirado. Por la forma en que se agregan y retiran elementos de la pila, el método ha sido llamado UEPS (Ultimas-Entradas, Primeras-Salidas), en inglés LIFO (Last-Input, First-Output).

OPERACIONES BASICAS

Las operaciones básicas que se pueden llevar a cabo en una pila son: Agregar en la pila, retirar de la pila y analizar si está vacía. Agregar (push) en la pila significa añadir un elemento a la pila. Dada una pila p y un elemento x, ejecutar la operación Agregar(p,x) es definido como añadir el elemento x por el tope de la pila p. La operación Retirar(p) retira (pop) el elemento tope de la pila p y regresa el valor del elemento, como valor de la función.

Debido a que la pila no puede ser infinita, esto es que el número de localidades es fijo, deberá verificarse el no exceder estos límites. Por esto, antes de agregar un nuevo elemento, hay que verificar si existe lugar para él, si no existe habrá un error en la operación de la pila, conocido como pila llena (overflow). De igual manera es necesario verificar en la operación de retiro si existe un elemento por retirar, si no habrá un error de pila vacía (underflow).

Para implementar la operación de retiro, debemos estar seguros que la pila no se encuentre vacía. La operación Vacío(P) determina si la pila p está vacía; si está vacía la operación regresa el valor verdadero, en caso contrario regresa el valor falso.

IMPLEMENTACION DE LA PILA Y SUS OPERACIONES

Una forma de implementar la pila se hace utilizando un tipo de datos contenido en el lenguaje PASCAL, éste es el ARRAY (arreglo) definido como una colección ordenada de elementos.

La diferencia fundamental entre la pila y el arreglo es la manera como se modifican y se accesan los elementos. El arreglo es de acceso aleatorio.

Una pila en PASCAL es declarada como un registro que contiene dos campos: Un arreglo para guardar los elementos de la pila y un entero para indicar la posición de la parte superior de la pila dentro del arreglo. Mostramos esta representación y las operaciones en una pila.

```
CONST
   maxpila=10;
TYPE
   pila=RECORD
           elemento:ARRAY[1..maxpila] OF pilatipo;
           tope:0..maxpila
        END;
VAR
   p:pila;
```

Asumimos que los elementos de la pila p contenidos en el arreglo llamado elemento son del tipo pilatipo y que la pila contiene a lo más maxpila elementos; en este ejemplo maxpila vale 10. Los elementos de la pila pueden ser enteros, reales, caracteres o cualquier otro tipo que se quiera definir. Por ejemplo: pilatipo.

Las operaciones básicas son:

Operación Agregar. Al implementar esta operación la posibilidad de pila llena debe ser tomada en cuenta. Cuando el arreglo está lleno es cuando la pila contiene tantos elementos como el arreglo, es decir cuando tope=maxpila, y un intento de agregar un nuevo elemento provoca que tope tome un valor fuera del rango de valores definidos, lo cual resulta un error.

Se introduce un procedimiento para ejecutar dicha operación:

```
PROCEDURE Agregar(VAR p:pila; x:pilatipo);
BEGIN
   p.tope:=p.tope+1;
   IF p.tope > maxpila THEN WRITELN('pila llena')
   ELSE  p.elem[p.tope]:=x;
END;
```

Pila vacía. La pila vacía no contiene elementos, lo que puede ser indicado por tope igual a 0. Escribimos una función que regrese verdadero si la pila está vacía y falso en caso contrario.

```
FUNCTION Vacio(p:pila):BOOLEAN;
BEGIN
   IF p.tope=0 THEN
      Vacio:=TRUE
   ELSE  Vacio:=FALSE
END;
```

Operación Retirar. Al implementar esta operación, la posibilidad de pila vacía debe ser tomada en cuenta. Se muestra una función que ejecuta dicha operación:

```
FUNCTION Retirar(VAR p:pila):pilatipo;
BEGIN
   IF Vacío(p) THEN WRITELN('pila vacía')
   ELSE
      BEGIN
         Retirar:=p.elem[p.tope];
         p.tope:=p.tope-1;
      END;
END;
```

Si la pila no está vacía, el elemento que se encuentra al tope es regresado como el valor de la función. Este elemento es retirado de la pila por la instrucción tope:=tope-1. Observamos que el valor de tope ha disminuido en uno, sin embargo el arreglo elemento sigue conteniendo este dato. De cualquier manera la pila ha cambiado, ya que contiene un elemento menos.

En seguida se presenta un programa para ilustrar las operaciones en una pila. En este ejemplo particular el arreglo es de 10 elementos y el tope de la pila toma los valores de 0 a 10. Inicialmente la pila contiene 6 elementos que se dan por el usuario en el momento de correr el programa, la pila se escribe a traves del procedimiento Escribe. El programa agrega otros 4 elementos y después retira 6 elementos de la pila. Se escribe nuevamente la pila después de agregar y retirar los elementos.

```
PROGRAM Operacion;
(Este programa ilustra las operaciones basicas en una pila)
CONST
   maxpila=10;
TYPE
   pilatipo=CHAR;
   pila=RECORD
            elem:ARRAY[1..maxpila] OF pilatipo;
            tope:0..maxpila
         END;
VAR
   a,b,c,d,e,f:CHAR;
   k:INTEGER;
   p:pila;

   (**************************************************************)
   PROCEDURE Escribe(p:pila);
   (se escriben los elementos de la pila y el valor del tope)
   BEGIN
      IF p.tope >=1 THEN
        FOR k:=p.tope DOWNTO 1 DO
          BEGIN
            WRITE(' ':14, p.elem[k]);
            IF k=p.tope THEN WRITELN('     p.tope=', p.tope:1)
            ELSE  WRITELN;
          END
      ELSE  WRITELN(' ':14,'      p.tope=',p.tope:1);
      WRITELN;   WRITELN;
   END;

   (Se omiten las siguientes rutinas, aparecen en la definicion
    de las operaciones basicas de la pila)
   (**************************************************************)
   PROCEDURE Agregar(VAR p:pila; x:pilatipo);
   (se agrega un elemento a la pila)
   (**************************************************************)
   FUNCTION Vacio(p:pila):BOOLEAN;
   (se prueba si la pila esta vacia)
   (**************************************************************)
   FUNCTION Retirar(VAR p:pila):pilatipo;
   (se retira el ultimo elemento de la pila)
   (**************************************************************)

                        ( Programa Principal )
BEGIN
   p.tope:=0;   (se inicializa la pila)
   WRITELN('            Pila Inicial             ');
   Escribe(p);
   (se agregan 6 elementos)
   Agregar(p,'a');
   Agregar(p,'b');
   Agregar(p,'c');
   Agregar(p,'d');
   Agregar(p,'e');
   Agregar(p,'f');
   WRITELN('    Pila con elementos agregados ');
```

```
    Escribe(p);
    (se retiran 4 elementos)
    f:=Retirar(p);
    e:=Retirar(p);
    d:=Retirar(p);
    c:=Retirar(p);
    WRITELN('   Pila con elementos retirados   ');
    Escribe(p);
END.
```

```
          Pila Inicial
                  p.tope=0


    Pila con elementos agregados
               f     p.tope=6
               e
               d
               c
               b
               a


   Pila con elementos retirados
               b     p.tope=2
               a
```

EJEMPLOS DE APLICACION DE LA PILA

Después de haber definido la pila y las operaciones sobre ella, mostraremos ahora cómo usar la pila en la solución de problemas.

a) Dada una expresión aritmética que incluye tres tipos de paréntesis ([],(),{}), probar que dichos paréntesis están correctamente anidados en la expresión.

Una pila es usada para guardar los paréntesis izquierdos encontrados; de acuerdo a como aparecen se van agregando en la pila. Cuando un paréntesis derecho es encontrado se retira el último elemento de la pila (un paréntesis izquierdo) y a través de una función llamada Compara se prueba si son del mismo tipo, esto significa que formen pareja de la misma clase; en caso de ser falso, la expresión aritmética es inválida.

La pila se escribe a través del procedimiento llamado Escribe cada vez que se agrega o retira un elemento de ella, para ir mostrando el procedimiento.

Vemos que la solución de este problema hace uso de una pila, ya que el último paréntesis en ser abierto debe ser el primero en ser cerrado. Esto es precisamente simulado por una pila, donde el último elemento en entrar es el primero en salir.

```
PROGRAM Parentesis;
{Este programa prueba que los parentesis que aparecen en una
 expresion aritmetica esten correctamente anidados}
CONST
   maxpila=10;      {numero maximo de elementos de la pila}
   maxc=50;         {numero maximo de caracteres de la expresion}
TYPE
   pilatipo=CHAR;  {tipo de elemento de la pila}
   Pila=RECORD
          elem:ARRAY[1..maxpila] OF CHAR;
          tope:0..maxpila
        END;
   cuerda=PACKED ARRAY[1..maxc] OF CHAR;
VAR
   ce:cuerda;       {guarda la cuerda de entrada o expresion}
   error:BOOLEAN; {verdadero cuando existe error en la ce}
   c:INTEGER;    {contador de caracteres de la  cuerda de entrada}
   i:pilatipo;  {guarda parentesis izquierdo retirado de la pila}
   p:pila;         {pila}

   {*********************************************************}
   PROCEDURE Escribe(p:pila);
   {se escriben la pila y el tope}
   VAR  k:INTEGER; {variable auxiliar}
   BEGIN
      IF p.tope=0 THEN
         WRITELN(' ':17,'   p.tope=',p.tope:1,'    pila vacia');
      FOR k:=p.tope DOWNTO 1 DO
         BEGIN
            WRITE(' ':16,p.elem[k]);
            IF k=p.tope THEN
               WRITELN('   p.tope=',p.tope:1)
            ELSE  WRITELN;
         END;
      WRITELN;  WRITELN;
   END;

   {Se omiten las siguientes rutinas, aparecen anteriormente}
   {*********************************************************}
   PROCEDURE Agregar(VAR p:pila; x:pilatipo);
   {*********************************************************}
   FUNCTION Vacio(p:pila):BOOLEAN;
   {*********************************************************}
   FUNCTION RETIRAR(VAR p:pila):pilatipo;
   {*********************************************************}

   FUNCTION Compara(i,d:pilatipo):BOOLEAN;
   {prueba si los parentesis izquierdo y derecho forman pareja}
   BEGIN
      IF ((i='(') AND (d=')')) OR
         ((i='{') AND (d='}')) OR
         ((i='[') AND (d=']')) THEN
         Compara:=TRUE
      ELSE
         Compara:=FALSE;
   END;
```

```
                          ( Programa Principal )
    BEGIN
    error:=FALSE;
    p.tope:=0;    (se inicializa el tope de la pila)
    c:=1;         (se inicia contador de caracteres de la expresion)
    WRITELN('Escribe la expresion aritmetica a probar:');
    READLN(ce);     WRITELN;
    WRITELN(' Se muestra como va quedando la pila de operadores');
    WRITELN;
    WHILE ce[c] <> '.' DO
       BEGIN
          CASE ce[c] OF
             '(','[','(':  BEGIN
                             Agregar(p,ce[c]); Escribe(p);
                           END;
             ')',']',')':  BEGIN
                              IF Vacio(p) THEN error:=TRUE
                              ELSE
                                BEGIN
                                  i:=Retirar(p);
                                  Escribe(p);
                                  IF NOT Compara(i,ce[c]) THEN
                                    error:=TRUE;
                                END;
                           END;
          END;
          c:=c+1;
       END;  WRITELN(ce);
    IF (NOT error) AND (vacio(p)) THEN
       WRITELN('Expresion aritmetica valida')
    ELSE
       WRITELN('Expresion aritmetica invalida');
END.
```

```
Escribe la expresion aritmetica a probar:
((a+b)-(-h])

Se muestra como va quedando la pila de operadores

                (    p.tope=1


                (    p.tope=2
                (


                (    p.tope=1


                     p.tope=0    pila vacia


((a+b)-(-h])
Expresion aritmetica invalida
```

```
Escribe la expresion aritmetica a probar:
(x+(y-[a+b])*c-(d+e))/(k-[m-n])

Se muestra como va quedando la pila de operadores

                (    p.tope=1


                (    p.tope=2
                (


                [    p.tope=3
                (
                (


                (    p.tope=2
                (


                (    p.tope=1


                (    p.tope=2
                (


                (    p.tope=1


                     p.tope=0   pila vacia


                (    p.tope=1


                [    p.tope=2
                (


                (    p.tope=1


                     p.tope=0   pila vacia

(x+(y-[a+b])*c-(d+e))/(k-[m-n])
Expresion aritmetica valida
```

```
PROGRAM Evaluar;
(Este programa evalua una expresion aritmetica sufija. La  expre-
 sion a evaluar es una cuerda de caracteres, que es convertida en
 enteros ('0'..'9') y operadores)
CONST
   maxc=80;  (numero maximo de caracteres de la expr. a evaluar)
TYPE
   expr=PACKED ARRAY[1..maxc] OF CHAR;
VAR
   ce:expr;  (expresion a evaluar)

   (****************************************************************)
   FUNCTION Eval(ce:expr):INTEGER;
   (recibe como entrada una expresion sufija y regresa el valor
    de la expresion)
   CONST
      maxpila=80;
   TYPE
      pilatipo=INTEGER;
      pila=RECORD
              elem:ARRAY[1..maxpila] OF pilatipo;
              tope:0..maxpila
           END;
   VAR
      opndp:pila;             (pila de operandos)
      i:1..maxc;              (indica posicion de la expresion)
      simb:CHAR;              (caracteres de la expresion aritmetica)
      valor:INTEGER;          (guarda el resultado de  una operacion)
      opnd1,opnd2:pilatipo;(operandos retirados de la pila)

      (****************************************************************)
      PROCEDURE Escribe(p:pila);
      (escribe el proceso de la evaluacion)
      VAR
         k:INTEGER;
      BEGIN
         IF simb IN ['0'..'9'] THEN
            WRITE(' ':43)
         ELSE
            BEGIN
               WRITE(opnd2:9);
               WRITE(opnd1:11);
               WRITE(valor:9,' ':14);
            END;
         FOR k:=1 TO p.tope DO
             WRITE(opndp.elem[k]:1,' ');
         WRITELN;
      END;

      (Se omiten las siguientes rutinas, se encuentran en la
       definicion de operaciones de la pila)
      (****************************************************************)
      PROCEDURE Agregar(VAR p:pila; x:pilatipo);
      (****************************************************************)
      FUNCTION Vacio(p:pila):BOOLEAN;
      (****************************************************************)
```

```
   FUNCTION RETIRAR(VAR p:pila):pilatipo;
   {**************************************************************}

   FUNCTION Opera(simb:CHAR; opnd1,opnd2:pilatipo):INTEGER;
   {de acuerdo al operador de entrada se ejecuta la operacion
   con los dos operandos y se regresa el valor}
   BEGIN
      IF simb IN ['+','*','-','/','$'] THEN
         CASE simb OF
            '+':    Opera:=opnd1+opnd2;
            '-':    Opera:=opnd1-opnd2;
            '*':    Opera:=opnd1*opnd2;
            '/':    Opera:=ROUND(opnd1/opnd2);
            '$':    Opera:=ROUND(EXP(opnd2*LN(opnd1)))
         END
      ELSE
         WRITELN('operador ilegal');
   END;
   {**************************************************************}
                              {Funcion Eval}
BEGIN
   i:=1;              {indica la posicion actual de ce}
   opndp.tope:=0;  {se inicializa el tope de la pila}
   WRITELN;
   WRITELN('simb      opnd2      opnd1      valor      opndp(pila)');
   simb:=ce[i];       {indica el simbolo corriente}
   {se analizan los simbolos de la expresion de entrada}
   WHILE simb <> ' ' DO
      BEGIN
         WRITE(' ':2,simb);
         IF simb IN ['0'..'9'] THEN
            BEGIN
               valor:=ORD(simb)-ORD('0');  {valor numerico del}
               Agregar(opndp,valor);                    {digito}
               Escribe(opndp);
            END
         ELSE
            BEGIN
               opnd2:=Retirar(opndp);
               opnd1:=Retirar(opndp);
               valor:=Opera(simb,opnd1,opnd2);
               Agregar(opndp,valor);
               Escribe(opndp)
            END;
         IF i < maxc THEN
            BEGIN
               i:=i+1;
               simb:=ce[i];
            END
         ELSE  simb:=' ';
      END;
   WRITELN;
   Eval:=Retirar(opndp);
   WRITE(' El resultado es ');
END;
{**************************************************************}
```

```
                  { Programa Principal}
BEGIN
   WRITELN('La expresion sufija a evaluar es: ');
   READLN(ce); WRITELN;
   WRITELN('Proceso de evaluacion de la expresion',Eval(ce));
END.
```

```
La expresion sufija a evaluar es:
623+-382/++2$3+

Proceso de evaluacion de la expresion
simb      opnd2     opnd1     valor      opndp(pila)
  6                                        6
  2                                        6 2
  3                                        6 2 3
  +         3         2         5          6 5
  -         5         6         1          1
  3                                        1 3
  8                                        1 3 8
  2                                        1 3 8 2
  /         2         8         4          1 3 4
  +         4         3         7          1 7
  *         7         1         7          7
  2                                        7 2
  $         2         7        49          49
  3                                        49 3
  +         3        49        52          52

 El resultado es 52
```

c) Manipulación de expresiones aritméticas.
Convertir una expresión infija en sufija.

Dada la expresión infija, que es una cuerda de caracteres, se analiza de izquierda a derecha. Si se encuentra con un operando, éste se inserta en la expresión sufija; si se encuentra un operador se analiza la pila; si ésta se halla vacía, el operador es agregado en la pila y en caso contrario se prueba a través de la función Prc la prioridad entre este operador presente y el operador tope de la pila. Si el operador tope es de mayor prioridad se retira de la pila agregándose en la expresión sufija y el operador presente se agrega en la pila.

En caso de que la expresión contenga paréntesis, cuando se encuentre un paréntesis izquierdo éste se agrega en la pila junto con los demás paréntesis izquierdos y operandos que existan, hasta encontrar un paréntesis derecho; éste no es agregado en la pila. En este momento se retiran los operandos que se encuentran entre la pareja de paréntesis y son puestos en la cuerda sufija. El paréntesis izquierdo se retira de la pila, se descarta y se sigue el proceso. Los paréntesis tienen mayor prioridad para su ejecución.

El programa principal lee la expresión infija y llama a la rutina Sufija, que es la encargada de hacer la conversión. Esta, a través de la rutina Escribe, imprime el proceso de la conversión mostrando cómo va quedando la cuerda sufija y cómo se va modificando la pila.

```
Program Convierte;
{Convierte una expresion aritmetica infija en sufija}
CONST
   maxexpr=80; {num. maximo de caracteres de la expr. aritmetica}
TYPE
   expr=PACKED ARRAY[1..maxexpr] OF CHAR;
VAR
   ce,cs:expr;   {cuerdas de entrada y salida, respectivamente}
   {*************************************************************}
   PROCEDURE IniciaLee(VAR ce,cs:expr);
   {inicializa las cuerdas y lee ce}
   VAR
      i:INTEGER;
   BEGIN
      FOR i:=1 TO maxexpr DO
         BEGIN
            ce[i]:=' ';  cs[i]:=' ';
         END;
      WRITELN('La expresion infija a convertir es:');
      READLN(ce);  WRITELN;
   END;
   {*************************************************************}
   PROCEDURE Sufija(ce:expr; VAR cs:expr);
   {convierte una expresion infija en sufija}
   CONST
      maxpila=20;   {maxima longitud de la pila}
   TYPE
      pilatipo=CHAR;
      pila=RECORD
              elem:array[1..maxpila] OF pilatipo;
              tope:0..maxpila
           END;
   VAR
      k:INTEGER;         {variable auxiliar}
      opdrp:pila;        {pila de operadores}
      j:1..maxexpr;      {indica posicion de la cuerda de salida}
      i:1..maxexpr;      {indica posicion de la cuerda de entrada}
      opdr, simb:CHAR;   {guardan operador y simbolo corriente}

      {Se omiten las siguientes rutinas, aparecen anteriormente}
      {**********************************************************}
      PROCEDURE Agregar(VAR p:pila; x:pilatipo);
      {**********************************************************}
      FUNCTION Vacio(p:pila):BOOLEAN;
      {**********************************************************}
      FUNCTION Retirar(VAR p:pila):pilatipo;
      {**********************************************************}

      FUNCTION Prcd(opdr1,opdr2:pilatipo):BOOLEAN;
      {asigna la prioridad de los operadores, es verdadera cuando
      la prioridad de opdr1 es mayor que opdr2}
      BEGIN
         CASE opdr1 OF
            '$','/','*':     IF opdr2 IN ['$','('] THEN
                                Prcd:=FALSE
                             ELSE
```

```
                           Prcd:=TRUE;
            '+','-':    IF opdr2 IN [')','-','+'] THEN
                           Prcd:=TRUE
                        ELSE
                           Prcd:=FALSE;
                '(':    Prcd:=FALSE;
                ')':    WRITELN('Error en parentesis')
      END; (del case)
   END;
   {*************************************************************}
   PROCEDURE Escribe;
   (escribe el simbolp, la cuerda de salida y la pila)
   BEGIN
      IF simb=' ' THEN  WRITE('    ')
      ELSE  WRITE('   ',simb);
      WRITE(' ':9);
      FOR k:=1 TO j DO  WRITE(cs[k]);
      IF NOT Vacio(opdrp) THEN
         BEGIN
            WRITE(' ':18-j);
            FOR k:=1 TO opdrp.tope DO  WRITE(opdrp.elem[k]);
            WRITELN;
         END
      ELSE WRITELN;
   END;
  (**************************************************************)
            (inicia procedimiento sufija)
BEGIN
   j:=0;              (indica la posicion actual de cs)
   i:=1;              (indica la posicion actual de ce)
   simb:=ce[i];       (indica el simbolo actual)
   opdrp.tope:=0;  (se inicializa el tope de la pila)
   WRITELN('  Proceso para convertir la expresion');
   WRITELN('simb       cuerda sufija      opdrp(pila)');
   (se analizan los simbolos hasta encontrar un blanco)
   WHILE simb <> ' ' DO
      BEGIN
         IF simb IN ['a'..'z','0'..'9'] THEN
            BEGIN
               j:=j+1;
               cs[j]:=simb;
            END
         ELSE
            BEGIN
               WHILE (NOT Vacio(opdrp))  AND
                   (Prcd(opdrp.elem[opdrp.tope],simb)) DO
                     BEGIN
                        opdr:=Retirar(opdrp);
                        j:=j+1;
                        cs[j]:=opdr;
                     END;
               IF Vacio(opdrp) OR (simb <> ')') THEN
                  Agregar(opdrp,simb)
               ELSE
                  opdr:=Retirar(opdrp);
            END;
```

```
            Escribe;
            IF i < maxexpr THEN
               BEGIN
                  i:=i+1;
                  simb:=ce[i];
               END
            ELSE
               simb:=' ';
         END;
      WHILE (NOT Vacio(opdrp)) DO
         BEGIN
            j:=j+1;
            cs[j]:=Retirar(opdrp);
            Escribe
         END;
      WRITELN;
      WRITE('Equivalente a notacion sufija es:   ');
      FOR j:=1 TO maxexpr DO WRITE(cs[j]); WRITELN;
   END;
   (*************************************************************)
                     {Programa principal}
BEGIN
   IniciayLee(ce,cs);
   Sufija(ce,cs);
END.
```

```
La expresion infija a convertir es:
((a-(b*c))+d)$(e+f)

Proceso para convertir la expresion
simb      cuerda sufija      opdrp(pila)
  (                          (
  (                          ((
  a         a                ((
  -         a                ((-
  (         a                ((-(
  b         ab               ((-(
  *         ab               ((-(*
  c         abc              ((-(*
  )         abc*             ((-
  )         abc*-            (
  +         abc*-            (+
  d         abc*-d           (+
  )         abc*-d+
  $         abc*-d+          $
  (         abc*-d+          $(
  e         abc*-d+e         $(
  +         abc*-d+e         $(+
  f         abc*-d+ef        $(+
  )         abc*-d+ef+       $
            abc*-d+ef+$

Equivalente a notacion sufija es:   abc*-d+ef+$
```

RECURSIVIDAD

El estudio de la recursividad se debe a los siguientes aspectos:

- Existen lenguajes no recursivos, como el FORTRAN y el COBOL.
- Se desea la comprensión del mecanismo de la recursividad. (implicaciones y dificultades).
- En ocasiones es necesario tener el control de la pila para poder hacer consultas o mejorar la eficiencia.

El lenguaje PASCAL permite al programador escribir procedimientos y funciones que se llaman a ellos mismos. Tales rutinas son llamadas recursivas. En la implementación de estas rutinas se hace uso de una pila, manejada por el sistema PASCAL. La pila es transparente para el usuario. Esta aplicación de la pila es una de las más importantes.

MECANISMO DE LA RECURSIVIDAD

En general cada vez que una rutina recursiva se llama a sí misma, una nueva área de datos particular debe ser asignada. Esta área de datos debe contener todos los parámetros, variables locales, variables temporales y la dirección de retorno. En recursión, esta área de datos está asociada a una llamada particular de la rutina. Cada llamada causa un área de datos nueva a ser asignada y solamente esta nueva área de datos puede ser referenciada dentro de tal llamada. Similarmente cada retorno de la rutina al punto donde la llamada fue hecha, causa que el área de datos actual sea liberada y que el área de datos que había sido asignada inmediatamente anterior se convierta en actual. Este comportamiento sugiere el uso de una pila.

Simulación de la Recursividad utilizando una Pila.

La pila se usa para guardar las generaciones sucesivas de variables y parámetros. Cada elemento de la pila es un área de datos conteniendo las variables y parámetros correspondientes a cada llamada, así como la dirección de regreso.

Cada vez que la rutina recursiva es llamada, una nueva área de datos es asignada y agregada en el tope de la pila. Los parámetros dentro de esta área de datos son inicializados para hacer referencia a los valores de sus correspondientes argumentos. La dirección de retorno dentro del área de datos es inicializada por la dirección siguiente de la instrucción de llamada. Cualquier referencia a variables o parámetros es hecha a través del tope actual (área de datos actual) de la pila. Cuando la rutina recursiva retorna, el valor de retorno (en caso de ser función) y la dirección de retorno son salvados y el área de datos es liberada; esto significa retirar el tope de la pila y hacer una transferencia de control a la dirección de retorno. La previa asignación es reactivada, es decir, viene a ser el tope actual de la pila.

A continuación presentamos dos algoritmos haciendo uso de la recursividad en PASCAL y sus correspondientes simulaciones.

d) Calcular el factorial de un número n.

```
PROGRAM Factorial;
{Se utiliza la recursividad del lenguaje.
 Calcula el factorial de un numero n, definido asi:
    Factorial(n)=1, si n=0
    Factorial(n)=n*Factorial(n-1), si n>0}
VAR
   n:INTEGER;

   {*************************************************************}
   FUNCTION Fact(n:INTEGER):INTEGER;
   {recibe como parametro de entrada un entero n y calcula su
    factorial}
   VAR
      x,y:INTEGER;
   BEGIN
      IF n < 0 THEN  WRITELN('Argumento negativo')
      ELSE
         IF n=0 THEN Fact:=1
         ELSE
            BEGIN
               x:=n-1;
               y:=Fact(x);
               Fact:=n*y;
            END;
   END;
   {*************************************************************}
                { Programa Principal }
BEGIN
   WRITE('Para que numero quieres el factorial?  ');
   READLN(n);
   WRITELN(' ':10,n,'!=',Fact(n));
END.
```

```
Para que numero quieres el factorial?  4
          4!=24
```

```
Para que numero quieres el factorial?  6
          6!=720
```

```
PROGRAM Factorial;
(Calcula el factorial de un numero n, simulando la recursividad
 utilizando una pila)
VAR
   n:INTEGER; (indica el numero a calcular su factorial)

   (**********************************************************)
   FUNCTION SimulaFact(n:INTEGER):INTEGER;
   (recibe como parametro de entrada un entero n y calcula su
    factorial, simulando la recursividad con el uso de una pila)
   CONST
      maxpila=80;
   TYPE
      pilatipo=INTEGER;
      pila=RECORD
              elem:ARRAY[1..maxpila] OF pilatipo;
              tope:0..maxpila
           END;
   VAR
      p:pila;
      k,x,y:INTEGER;

      (Se omiten las siguientes rutinas)
      (*******************************************************)
      PROCEDURE Agregar(VAR p:pila; x:pilatipo);
      (*******************************************************)
      FUNCTION Vacio(p:pila):BOOLEAN;
      (*******************************************************)
      FUNCTION Retirar(VAR p:pila):pilatipo;
      (*******************************************************)

      PROCEDURE Escribe(p:pila; y:INTEGER);
      (se escriben los elementos de la pila y el entero 'y' que
      va  acumulando el resultado  final ('y' es el producto de
      'y' con el elemento tope de la pila, inicialmente y=1))
      BEGIN
         IF NOT Vacio(p) THEN
            FOR k:=p.tope DOWNTO 1 DO
               WRITELN(' ':9,p.elem[k])
         ELSE  WRITELN;
         WRITE(' ':5,'p (pila)');
         WRITE(' ':5,'y=',y);
         WRITELN;  WRITELN;
      END;
      (*******************************************************)
                          (inicia SimulaFact)
   BEGIN
      p.tope:=0;
      x:=n;
      IF n < 0  THEN
         WRITELN('argumento negativo')
      ELSE
         WHILE x <> 0 DO
            BEGIN
               Agregar(p,x);
               x:=x-1;
```

```
            END;
      y:=1;
      Escribe(p,y);
      WHILE (NOT Vacio(p)) DO
         BEGIN
            x:=Retirar(p);
            y:=x*y;     {se va acumulando resultado final}
            Escribe(p,y);
         END;
      SimulaFact:=y;
      WRITE('El factorial de ',n,' es  ');
   END;
   {*************************************************************}
                   { Programa Principal }
BEGIN
   WRITE('Para que numero quieres el factorial? '); READLN(n);
   WRITELN('Proceso para calcular el factorial'); WRITELN;
   WRITELN(SimulaFact(n));
END.
```

```
Para que numero quieres el factorial? 5
Proceso para calcular el factorial

         1
         2
         3
         4
         5
     p (pila)      y=1

         2
         3
         4
         5
     p (pila)      y=1

         3
         4
         5
     p (pila)      y=2

         4
         5
     p (pila)      y=6

         5
     p (pila)      y=24

     p (pila)      y=120

El factorial de 5 es  120
```

e) Permutaciones.

Calcular las n! permutaciones de n elementos a1,..,an guardados en un arreglo, sobre la misma área de memoria sin recurrir a otro arreglo.

De igual manera presentamos este ejercicio usando la recursividad de PASCAL y por otro lado simulándola; en este último se han considerado todas las variables locales, parámetros y las direcciones de retorno, dentro del área de datos citada anteriormente, en este caso llamada dato. Se define un área de datos tipo dato, de tal manera que esta represente al dato actual, es decir el elemento tope de la pila, con el fin de facilitar el análisis de la pila.

Los elementos (datos) de la pila son registros y en PASCAL las funciones no pueden regresar un registro, por lo tanto no podemos utilizar la función _Retirar_ que hemos venido usando. En su lugar definimos un procedimiento.

```
PROGRAM Permuta;
{Se utiliza la recursividad del lenguaje.
 Calcula las n! permutaciones de n elementos}
CONST
   maxc=10;
TYPE
   cuerda=PACKED ARRAY[1..maxc] OF CHAR;
VAR
   a:cuerda;
   n:INTEGER;

   {**********************************************************}
   PROCEDURE Per(a:cuerda; n:INTEGER);
   {recibe un arreglo de caracteres y encuentra todas las permuta-
    ciones de los elementos del arreglo}
   VAR
      i:INTEGER;
      k:CHAR;
   BEGIN
      IF n=1 THEN  WRITELN(' ':9,a)
      ELSE
         BEGIN
            Per(a,n-1);
            FOR i:=1 TO n-1 DO
               BEGIN
                  k:=a[i];
                  a[i]:=a[n];
                  a[n]:=k;
                  Per(a,n-1);
               END;
         END;
   END;
   {**********************************************************}
                  { Programa Principal }
BEGIN
   WRITE('Cuantos elementos quieres permutar? ');
   READLN(n);
   WRITE('Cuales son? ');
   READLN(a);
   WRITELN('Las permutaciones son: ');
   Per(a,n);
END.
```

```
Cuantos elementos quieres permutar? 3
Cuales son? abc
Las permutaciones son:
         abc
         bac
         cba
         bca
         cab
         acb
```

```
PROGRAM SimulaPermutacion;
(Calcula las n' permutaciones de n elementos, simulando la
 recursividad con el uso de la pila)
CONST
   maxc=10;
TYPE
   cuerda=PACKED ARRAY[1..maxc] OF CHAR;
VAR
   a:cuerda; n:INTEGER;

   (*************************************************************)
   PROCEDURE Simulaper(a:cuerda; n:INTEGER);
   LABEL
      1,2,3,10;
   CONST
      maxpila=50;
   TYPE
      indice=0..maxc;
      dato=RECORD
                cadena:cuerda;
                num,i:INDICE;
                x:CHAR;
                regreso:1..3
             END;
      pila=RECORD
                elem:ARRAY[1..maxpila] OF dato;
                tope:0..maxpila
             END;
   VAR
      p:pila;
      datoact:dato;
      r:1..3;
      (*************************************************************)
      PROCEDURE Agregar(VAR p:pila; datoact:dato);
      (se agrega un elemento a la pila)
      BEGIN
         p.tope:=p.tope+1;
         IF p.tope > maxpila THEN WRITELN('pila llena')
         ELSE  p.elem[p.tope]:=datoact;
      END;
      (*************************************************************)
      FUNCTION Vacio(p:pila):BOOLEAN;
      (se prueba si la pila esta vacia)
      BEGIN
         IF p.tope=0 THEN  Vacio:=TRUE
         ELSE  Vacio:=FALSE;
      END;
      (*************************************************************)
      PROCEDURE Retirar(VAR p:pila; VAR datoact:dato);
      (se retira el ultimo elemento de la pila)
      BEGIN
         IF Vacio(p) THEN WRITELN('Pila VAcia')
         ELSE BEGIN
                 datoact:=p.elem[p.tope]; p.tope:=p.tope-1;
               END;
      END;
```

```
    {***********************************************************}
                    {inicia SimulaPer}
BEGIN
     p.tope:=0;  {inicializacion de variables}
     datoact.cadena:=a;
     datoact.num:=n;
     datoact.i:=0;
     datoact.k:=' ';
     datoact.regreso:=3;
     10:    IF datoact.num=1 THEN
                WRITELN(' ':9,datoact.cadena)
            ELSE
                BEGIN
                   {se prepara para hacer un llamado}
                   {se actualizan las variables que cambian}
                   Agregar(p, datoact);
                   datoact.num:=datoact.num-1;
                   datoact.regreso:=1;
                   GOTO 10;
                   1:    datoact.i:=1;
                         WHILE datoact.i < datoact.num DO
                         BEGIN
                            datoact.k:=datoact.cadena[datoact.i];
                            datoact.cadena[datoact.i]:=
                            datoact.cadena[datoact.num];
                            datoact.cadena[datoact.num]:=datoact.k;
                          {se prepara para hacer un llamado}
                          {se actualizan las variables que cambian}
                            Agregar(p, datoact);
                            datoact.num:=datoact.num-1;
                            datoact.regreso:=2;
                            GOTO 10;
                            2:    datoact.i:=datoact.i+1;
                         END;
                END;
                {simular el regreso}
                r:=datoact.regreso;
                IF NOT Vacio(p) THEN  Retirar(p, datoact);
                CASE r OF
                   1:    GOTO 1;
                   2:    GOTO 2;
                   3:    GOTO 3;
                END;
                3:
   end;
   {***********************************************************}
                 {Programa Principal}
BEGIN
   WRITE('Cuantos elementos quieres permutar? '); READLN(n);
   WRITE('Cuales son? '); READLN(a);
   WRITELN('Las permutaciones son:');
   SimulaPer(a,n);
END.
```

```
Cuartos elementos quieres permutar? 4
Cuales son? abcd
Las permutaciones son:
          abcd
          bacd
          cbad
          bcad
          cabd
          acbd
          dbca
          bdca
          cbda
          bcda
          cdba
          dcba
          dacb
          adcb
          cadb
          acdb
          cdab
          dcab
          dabc
          adbc
          badc
          abdc
          bdac
          dbac
```

COLAS

Esta estructura de datos frecuentemente se utiliza para simular situaciones del mundo real.

DEFINICION

Una **cola** es una colección ordenada de elementos a partir de la cual se pueden eliminar elementos de un extremo (llamado el frente de la pila) y en la cual tambien se pueden agregar elementos en el otro extremo (llamado fondo de la cola).

CARACTERISTICA PRINCIPAL

El primer elemento que se adiciona a una cola es el primero en ser retirado. Por esta razón una cola es algunas veces denominada PEPS (Primeros en Entrar, Primeros en Salir) en ingles FIFO (First Input, First Output).

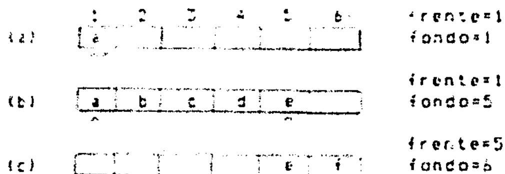
OPERACIONES BASICAS

Existen cuatro operaciones básicas que pueden aplicarse a una cola. La operación Agregar(q,x) adiciona el elemento x por el fondo de la cola q. Retirar(q) elimina el elemento del frente de la cola. Vacia(q) retorna el valor de falso o verdadero dependiendo si la cola contiene algún elemento o esta vacía. La última operación, Llena(q) regresa el valor de verdadero o falso ya sea que la cola este llena o no.

REPRESENTACION DE LA COLA EN PASCAL

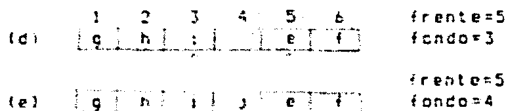
Se utiliza un arreglo que contenga los elementos de la cola y dos variables, llamadas frente y fondo, las cuales contienen las posiciones del arreglo, correspondientes al primero y al último elementos de la cola.

Examinemos que sucede al utilizar esta representación. Asumimos que el máximo número de elementos de la cola es seis (maxcola=6). En la figura (a) se ha agregado un elemento a la cola. En la figura (b) se han agregado otros 4 elementos. En (c) se han retirado 4 y se ha agregado otro.



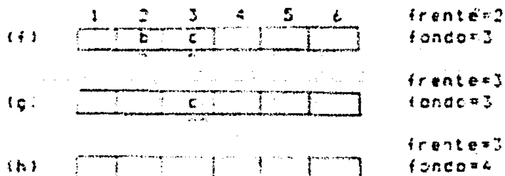
Observamos que en la cola solamente existen dos elementos, entonces debe haber espacio para que la cola se expanda sin temor a un sobreflujo. No obstante para insertar un elemento más en la cola, la variable fondo debe ser incrementada, tomando el valor de 7, pero el arreglo es de solamente seis elementos, entonces no se puede hacer la inserción de este nuevo elemento. De aquí que es posible llegar a tener la cola vacía, sin embargo no se pueda agregar un nuevo elemento.

Una solución a este problema es considerar el arreglo que contiene la cola como un círculo en lugar de una línea recta (ver la siguiente figura). Es decir, que nos podemos imaginar que el primer elemento del arreglo está inmediatamente después del último elemento. Esto implica que aun si el lugar del último elemento está ocupado, puede insertarse un nuevo valor detrás de este como primer elemento del arreglo siempre y cuando el lugar del primer elemento esté vacío.



Partiendo de la estructura cola de la figura (c) (hoja anterior), han sido agregados otros 3 elementos a la cola, figura (d), posteriormente otro elemento, figura (e). Observamos que después de esta operación de agregar, la cola ha sido llenada con la condición frente=fondo+1.

Analizamos ahora el caso de cola vacía.



Asumamos que la cola contiene dos elementos, figura (f), posteriormente los dos elementos son retirados de la cola, figuras (g) y (h), quedando ésta vacía después de la operación de retirar con la condición frente=fondo+1.

Sucede que en los dos casos, tanto cuando la cola está llena o cuando la cola está vacía se cumple la condición frente=fondo+1.

Una forma de resolver este problema es agregar a nuestra estructura de cola una variable booleana que nos recuerde cual fue la última operación que hicimos (llenar o vaciar).

Rutinas de las operaciones básicas en una cola.

```
TYPE
   cola=RECORD
           elem:ARRAY[1..maxcola] OF colatipo;
           frente,fondo:0..maxcola;
           llenando:BOOLEAN
        END;
```

```
FUNCTION Vacio(q:cola):BOOLEAN;
(es verdadera si la cola está vacía)
BEGIN
   Vacío:=(NOT q.llenando) AND (q.frente=(q.fondo+1) MOD
            maxcola);
END;


FUNCTION Lleno(q:cola):BOOLEAN;
(es verdadera si la cola está llena)
BEGIN
   Lleno:=q.llenando AND (q.frente=(q.fondo+1) MOD maxcola);
END;

PROCEDURE Agregar(VAR q:cola; x:colatipo);
(agrega el elemento x por el fondo de la pila)
BEGIN
   IF NOT Lleno(q) THEN
      BEGIN
         IF q.fondo=maxcola THEN q.fondo:=1
         ELSE  q.fondo:=q.fondo+1;
         q.elem[q.fondo]:=x;
         q.llenando:=TRUE;
      END
   ELSE
      WRITELN('Cola llena');
END;

FUNCTION Retirar(VAR q:cola):colatipo;
(retira el elemento que está al frente de la cola)
BEGIN
   IF NOT Vacío(q) THEN
      BEGIN
         Retirar:=q.elem[q.frente];
         IF q.frente=maxcola THEN q.frente:=1
         ELSE
            q.frente:=q.frente+1;
         q.llenando:=FALSE;
      END
   ELSE   WRITELN('Cola vacía);
END;
```

En seguida se presenta un programa para ilustrar las operaciones en una cola.

Los ejemplos de aplicación de colas aparecen posteriormente, debido a que en estas aplicaciones se usan otras estructuras de datos, como son: listas y arboles.
En el tema de listas páginas 48-52 presentamos el problema de simulación y en el tema de arboles páginas 85-98 el de juegos.

```
PROGRAM Operacion;
{ Este programa ilustra las operaciones basicas en una cola, im-
 plementada de manera circular}
CONST
   maxcola=6; {longitud maxima de la cola}
TYPE
   colatipo=CHAR;
   cola=RECORD
           elem:ARRAY[1..maxcola] OF colatipo;
           frente,fondo:0..maxcola;
           llenando:BOOLEAN
        END;
VAR
   q:cola;
   k:INTEGER;
   a,b,c,d,e,f,g:CHAR;

   {************************************************************}
   PROCEDURE Inicia;
   {se inicializa la cola}
   BEGIN
      FOR k:=1 TO maxcola DO q.elem[k]:=' ';
      q.llenando:=FALSE;
      q.frente:=0;
      q.fondo:=0;
   END;

   {Se omiten las siguientes rutinas}
   {************************************************************}
   FUNCTION Vacio(q:cola):BOOLEAN;
   {es verdadera si la cola esta vacia}
   {************************************************************}
   FUNCTION Lleno(q:cola):BOOLEAN;
   {es verdadera si la cola esta llena}
   {************************************************************}
   PROCEDURE Agregar(VAR q:cola; x:colatipo);
   {agrega el elemento x por el fondo de la cola}
   {************************************************************}
   FUNCTION Retirar(VAR q:cola):colatipo;
   {retira el elemento x por el fondo de la cola}
   {************************************************************}

   PROCEDURE Escribe(q:cola);
   {se escriben los elementos de la cola y las respectivas posi-
    ciones de las variables llamadas frente y fondo}
   BEGIN
      IF q.frente=1 THEN
         BEGIN
            WRITE(' ':8);
            FOR k:=1 TO maxcola DO  WRITE(k); WRITELN;
         END;
      WRITE(' ':8);
      FOR k:=1 TO maxcola DO  WRITE(q.elem[k]); WRITELN;
      WRITELN(' ':20,'frente=',q.frente);
      WRITELN(' ':21,'fondo=',q.fondo); WRITELN;
   END;
```

```
   (***********************************************************)
                  { Programa Principal}
BEGIN
   Inicia;
   {se agregan cinco elementos en la cola}
   Agregar(q,'a'); q.frente:=1;
   Agregar(q,'b');
   Agregar(q,'c');
   Agregar(q,'d');
   Agregar(q,'e');
   WRITELN('        Cola inicial');
   Escribe(q);
   a:=Retirar(q);
   b:=Retirar(q);
   c:=Retirar(q);
   WRITELN('Cola con tres elementos retirados');
   Escribe(q);
   Agregar(q,'f');
   Agregar(q,'g');
   WRITELN('Se agregan otros dos elementos a la cola');
   Escribe(q);
END.
```

```
        Cola inicial
          123456
          abcde
          ^   ^
                   frente=1
                    fondo=5

Cola con tres elementos retirados
          abcde
             ^^
                   frente=4
                    fondo=5

Se agregan otros dos elementos a la cola
          gbcdef
          ^  ^
                   frente=4
                    fondo=1
```

LISTAS

Hasta este momento hemos analizado dos estructuras de datos que son la PILA y la COLA, en las cuales solamente podemos insertar o remover el elemento que está al principio o final de la estructura; estas operaciones no se pueden hacer en lugares arbitrarios. La estructura que nos resuelve esta situacion es la lista. Es utilizada cuando se presentan los casos siguientes:

- Existen frecuentes inserciones o remociones en lugares arbitrarios de la estructura.
- El tamaño de la estructura a manejar no se conoce con anticipación o la cota máxima puede resultar muy alta.

DEFINICION

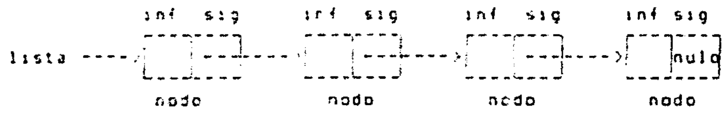
La **lista** es un conjunto ordenado de nodos o elementos, los cuales pueden variar a medida que los nodos son insertados o removidos desde lugares arbitrarios de la estructura.

La lista puede ser representada de manera secuencial o ligada en la memoria de la máquina. La diferencia entre representación secuencial y ligada es que en la segunda el orden se da explícitamente por medio de un campo liga o apuntador.

Las listas ligadas en estructura de datos nos dan mucha flexibilidad para su manipulación. Todas las demas estructuras de datos se pueden representar usando listas ligadas, en particular la PILA y la COLA, ya tratadas con representación secuencial. Más adelante daremos las representaciones y operaciones basicas con listas ligadas.

La forma para representar una lista lineal ligada es expandir cada nodo para contener el campo liga o apuntador al siguiente elemento o nodo sucesor.

Se muestra en la siguiente figura:



Cada nodo de la lista contiene dos campos, un campo de información inf y un campo sig que indica la dirección del elemento siguiente de la lista. Esta dirección, la cual es utilizada para dar acceso a un nodo en particular, es denominada apuntador. La lista ligada completa es accesada desde un apuntador externo de la lista llamada lista que apunta al primer nodo de la lista, conteniendo su dirección. El campo de dirección siguiente del último nodo en la lista contiene un valor especial llamado nulo. Este apuntador nulo es empleado para indicar el fin de la lista. La lista sin nodos se denomina lista vacía y el valor del apuntador externo de la lista es el apuntador nulo.

CARACTERISTICA PRINCIPAL

Los nodos en una lista pueden ser insertados o retirados desde lugares arbitrarios de la lista.

OPERACIONES BASICAS

Consideramos las cuatro operaciones siguientes como básicas, ya que a partir de ellas se pueden construir otras según las necesidades de un programa.

Insertacabeza(lista, x), inserta un nodo con información x por la cabeza de la lista apuntada por la variable lista.

Insertadespués(p, x), inserta un nodo con información x justo después del nodo apuntado por p.

Retiracabeza(lista, x), retira un nodo por la cabeza de la lista apuntada por lista, guardando la información del nodo en x.

Retiradespués(p, x), retira el nodo que se encuentra justo después del nodo apuntado por p, guardando la información en x.

IMPLEMENTACION DE LISTAS LIGADAS EN PASCAL

El número de nodos en una lista puede variar a medida que los nodos son insertados o retirados; por lo tanto deben existir mecanismos para obtener nodos disponibles y para liberar nodos.

Existen dos formas de implementación, que son a través de un arreglo o de estructuras dinámicas. Presentaremos las dos maneras de implementación. (En las páginas 52 y 53 trataremos la implementación dinámica.)

IMPLEMENTACION CON ESTRUCTURAS ESTATICAS

La implementación de listas a través de un arreglo o estructuras estáticas puede ser declarada como sigue:

```
CONST
   numnodos=  ;
TYPE
   apunt:=0..numnodos;
   nodotipo=RECORD
               inf: inftipo;
               sig:apunt
            END;
VAR
   espacio: ARRAY[1..numnodos] OF nodotipo;
```

En este esquema, un apuntador a un nodo es un entero entre 1 y numnodos; hace referencia a un elemento en particular del arreglo espacio. El apuntador nulo se representa por el entero 0.

Inicialmente todos los nodos están sin uso, puesto que no se ha formado aún ninguna lista; por lo tanto todos se encuentran formando una lista de nodos disponibles, como se muestra:

```
PROCEDURE Inicia;
CONST
    nulo=0;
VAR
   i:INTEGER;
BEGIN
   FOR i:=1 TO numnodos-1 DO
      espacio[i].sig:=i+1;
   espacio[numnodos].sig:=nulo;
   disponible:=1;
END;
```

Cuando se requiere un nodo para ser usado en una lista en particular, se obtiene de la lista de nodos disponibles. Igualmente cuando ya no se necesita un nodo, éste regresa a la lista de nodos disponibles.

Estas dos operaciones son implantadas por las rutinas Traenodo y Liberanodo.

```
PROCEDURE Traenodo(p:apunt);
{libera  un nodo de la lista de nodos disponibles, apuntándolo
 por p}
BEGIN
   IF disponible=nulo THEN
      WRITELN('No existen nodos disponibles')
   ELSE
      BEGIN
         p:=disponible;
         disponible:=espacio[disponible].sig;
     END;
END;

PROCEDURE Liberanodo(p:apunt);
{el nodo p es agregado a la lista de nodos disponibles}
BEGIN
   espacio[p].sig:=disponible;
   disponible:=p;
END;
```

Las operaciones básicas son:

```
PROCEDURE Insertacabeza(VAR lista:apunt; x:inftipo);
{inserta un  nodo con informacion x a la lista cuya cabeza es
 apuntada por la variable llamada lista}
VAR  q:apunt;
BEGIN
   Traenodo(q);
   espacio[q].inf:=x;
   espacio[q].sig:=lista;
   lista:=q;
END;

PROCEDURE Insertadespues(p:apunt; x:inftipo);
{inserta un nodo despues del nodo apuntado por p}
VAR  q:apunt;
BEGIN
   IF p=nulo THEN
      WRITELN('inserción de un vacío')
   ELSE
      BEGIN
         Traenodo(q);
         espacio[q].inf:=x;
         espacio[q].sig:=espacio[p].sig;
         espacio[p].sig:=q;
      END;
END;
```

```
PROCEDURE Retiracabeza(VAR lista:apunt; x:inftipo);
(retira un nodo de la cabeza de la lista apuntada por lista)
VAR
   q:apunt;
BEGIN
   IF lista=@ THEN  WRITELN('lista vacía')
   ELSE
      BEGIN
         q:=lista;
         x:=q^.inf;
         lista:=espacio[lista].sig;
         Liberanodo(q);
      END;
END;

PROCEDURE Retiradespués(p:apunt; x:inftipo);
(retira el nodo colocado después del que apunta p)
Var
   q:apunt;
BEGIN
   IF (p=nulo) OR (espacio[p].sig=nulo) THEN
      WRITELN('eliminación de un vacío')
   ELSE
      BEGIN
         q:=espacio[p].sig;
         x:=q^.inf;
         espacio[p].sig:=espacio[q].sig;
         Liberanodo(q);
      END;
END;
```

En seguida se presenta un programa para ilustrar las operaciones en una lista.

```
PROGRAM Operacion;
(Este programa ilustra las operaciones para la manipulacion de
 listas, cuya implementacion se basa en estructuras estaticas)
CONST
   numnodos=50;    (numero de nodos disponibles)
   nulo=0;         (apuntador nulo)
TYPE
   inftipo=INTEGER;
   apunt=0..numnodos;
   nodotipo=RECORD
      inf:inftipo;
      sig:apunt
   END;
VAR
   espacio:ARRAY[1..numnodos] OF nodotipo;(espacio de nodos)
   entero:ARRAY[1..numnodos] OF INTEGER;  (elementos para listas)
   disponible,l1,l2,l3:apunt;             (apuntadores)
   laux,laux1,laux2,laux3:apunt;
   n1,n2,n3,long,i,ent:INTEGER;           (variables auxiliares)
   seleccion: INTEGER;

   (*********************************************************)
   PROCEDURE Inicializa;
   (inicializa el espacio disponible ligando cada nodo con el
    siguiente, el ultimo apunta a nulo, ademas inicializa todas
    las listas utilizadas)
   VAR  j:INTEGER;
   BEGIN
      FOR j:=1 TO numnodos-1 DO
         espacio[j].sig:=j+1;
      espacio[numnodos].sig:=nulo; disponible:=1;
      l1:=nulo; l2:=nulo; l3:=nulo;
      laux:=nulo; laux1:=nulo; laux2:=nulo; laux3:=nulo;
   END;

   (Se omiten las operaciones basicas, aparecen anteriormente)
   (*********************************************************)
   PROCEDURE Traenodo(VAR p:apunt);
   (libera un nodo de la lista del espacio disponible, apuntandolo
    por p)
   (*********************************************************)
   PROCEDURE Liberanodo(p:apunt);
   (el nodo p es agregado a la lista de nodos disponibles)
   (*********************************************************)
   PROCEDURE Insertacabeza(VAR lista:apunt; x:inftipo);
   (inserta un nodo con informacion x a la lista cuya cabeza es
    apuntada por la variable llamada lista)
   (*********************************************************)
   PROCEDURE Insertadespues(p:apunt; x:inftipo);
   (inserta un nodo despues del nodo apuntado por p)
   (*********************************************************)
   PROCEDURE Retiracabeza(VAR lista:apunt; VAR x:inftipo);
   (retira un nodo de la cabeza de la lista apuntada por lista)
   (*********************************************************)
   PROCEDURE Retiradespues(p:apunt; VAR x:inftipo);
   (retira un nodo colocado despues del que apunta p)
```

```
{****************************************************************}
PROCEDURE Insertafinal(VAR lista:apunt; x:inftipo);
{inserta al final de la lista un nodo con informacion x}
VAR
   p,q:apunt;
BEGIN
   p:=nulo;
   q:=lista;
   WHILE q <> nulo DO
    BEGIN
      p:=q;  q:=espacio[q].sig;
    END;
   IF p=nulo THEN
      Insertacabeza(lista,x)
   ELSE
      Insertadespues(p,x);
END;
{****************************************************************}
PROCEDURE Insertaiesimo(VAR lista:apunt; i:INTEGER; x:inftipo);
{inserta un nodo con informacion x en el lugar iesimo}
VAR
   p,q:apunt;
   ie:INTEGER;
BEGIN
   p:=nulo;
   q:=lista;
   ie:=i;
   WHILE (q <> nulo) AND (i <> 1) DO
   BEGIN
      i:=i-1;
      p:=q; q:=espacio[q].sig;
   END;
   IF q=nulo THEN WRITELN('no existe el lugar ',ie)
   ELSE
      IF p=nulo THEN Insertacabeza(lista,x)
      ELSE
         Insertadespues(p,x);
END;
{****************************************************************}
PROCEDURE Retirafinal(VAR lista:apunt; x:inftipo);
{retira un nodo al final de la lista apuntada por lista, guar-
 dando su informacion en x}
VAR
   p,q:apunt;
BEGIN
   p:=nulo;
   q:=lista;
   WHILE espacio[q].sig <> nulo DO
   BEGIN
      p:=q; q:=espacio[q].sig;
   END;
   IF p=nulo THEN
      WRITELN('se retira al final de una lista vacia')
   ELSE
      Retiradespues(p,x);
END;
```

```
(*************************************************************)
PROCEDURE Retiraiesimo(VAR lista:apunt; i:INTEGER; x:inftipo);
(retira el nodo que se encuentra en el lugar iesimo de la lis-
 ta, guardando su informacion en x)
VAR
   p,q:apunt;
   ie:INTEGER;
BEGIN
   p:=nulo;
   q:=lista;
   ie:=i;
   WHILE (q <> nulo) AND (i <> 1) DO
   BEGIN
      i:=i-1;
      p:=q; q:=espacio[q].sig;
   END;
   IF q=nulo THEN
      WRITELN('no existe el lugar ',ie,' en la lista')
   ELSE
      IF p=nulo THEN Retiracabeza(lista,x)
      ELSE
         Retiradespues(p,x);
END;
(*************************************************************)
PROCEDURE Liberatodos(VAR lista:apunt);
(libera todos los nodos de la lista apuntada por lista)
VAR
   q:apunt;
BEGIN
   WHILE lista <> nulo DO
   BEGIN
      q:=lista;
      lista:=espacio[lista].sig;
      Liberanodo(q);
   END;
   WRITELN('la lista queda vacia');
   WRITELN; WRITELN;
END;
(*************************************************************)
Function Longitud(lista:apunt):INTEGER;
(calcula la longitud de la lista apuntada por lista)
VAR
   q:apunt;
   aux:INTEGER;
BEGIN
   aux:=0;
   q:=lista;
   WHILE q <> nulo DO
   BEGIN
      aux:=aux + 1;
      q:=espacio[q].sig;
   END;
   Longitud:=aux;
   WRITELN('su longitud es  ',aux);
   WRITELN;
END;
```

```
{***************************************************************}
PROCEDURE Invierte(VAR lista:apunt);
{invierte una lista, de tal manera que el ultimo elemento pasa
 a ser el primero y, asi sucesivamente}
VAR
   l,q:apunt;
BEGIN
   q:=lista;  l:=nulo;
   IF q=nulo THEN WRITELN('la lista esta vacia')
   ELSE
      WHILE q <> nulo DO
         BEGIN
            Insertacabeza(l,espacio[q].inf);
            q:=espacio[q].sig;
         END;
   lista:=l;
END;
{***************************************************************}
PROCEDURE Colocaordenando(VAR lista:apunt; x:inftipo);
{coloca un nodo con informacion x en el lugar correspondiente
 en orden creciente de la lista apuntada por lista}
VAR
   p,q:apunt;
BEGIN
   p:=nulo;   q:=lista;
   WHILE (q <> nulo) AND (espacio[q].inf <= x) DO
   BEGIN
      p:=q;
      q:=espacio[q].sig;
   END;
   IF p=nulo THEN
      Insertacabeza(lista,x)
   ELSE
      Insertadespues(p,x);
END;
{***************************************************************}
Function Concatena(lista1,lista2:apunt):apunt;
{regresa un apuntador a la nueva lista formada con los elemen-
 tos de la lista apuntada por lista1 seguidos de lista2}
VAR
   p,r:apunt;
   i:1..2;
BEGIN
   r:=nulo;
   FOR i:=1 TO 2 DO
      BEGIN
         IF i=1 THEN p:=lista1 ELSE p:=lista2;
         WHILE p <> nulo DO
            BEGIN
               IF r=nulo THEN
                  BEGIN
                     Insertacabeza(r,espacio[p].inf);
                     Concatena:=r;
                  END
               ELSE
                  BEGIN
```

```
                        Insertadespues(r,espacio[p].inf);
                        r:=espacio[r].sig;
                     END;
                  p:=espacio[p].sig;
               END;
         END;
END;
(************************************************************)
Function Combinaordenando(lista1,lista2:apunt):apunt;
(regresa un apuntador a la nueva lista que resulta de combinar
 en orden creciente las listas apuntadas por lista1 y lista2,
 ya ordenadas en forma creciente)
VAR
   p,q,r,t,aux:apunt;
   ter:inftipo;
BEGIN
   p:=lista1;
   q:=lista2;
   Traenodo(t);       (nodo auxiliar que al final se retira)
   espacio[t].sig:=0;
   Combinaordenando:=t;   r:=t;
   WHILE (p <> nulo) AND (q <> nulo) DO
      BEGIN
         IF espacio[p].inf <= espacio[q].inf THEN
            BEGIN
               ter:=espacio[p].inf;
               p:=espacio[p].sig;
            END
         ELSE
            BEGIN
               ter:=espacio[q].inf;
               q:=espacio[q].sig;
            END;
         insertadespues(r,ter);
         r:=espacio[r].sig;
      END;
      IF p <> nulo THEN
         aux:=p
      ELSE
         aux:=q;
      WHILE aux <> nulo DO
         BEGIN
            Insertadespues(r,espacio[aux].inf);
            aux:=espacio[aux].sig;
            r:=espacio[r].sig;
         END;
      Combinaordenando:=espacio[t].sig;
      Liberanodo(t);      (se retira nodo auxiliar)
END;
(************************************************************)
PROCEDURE EscribeLista(lista:apunt);
(imprime el campo inf. de cada nodo de la lista)
BEGIN
   IF lista=nulo THEN WRITELN('lista vacia')
   ELSE
      repeat
```

```
            WRITE(espacio[lista].inf,'--->');
            lista:=espacio[lista].sig;
         until lista=nulo;
      WRITELN;
   END;
   {**************************************************************}
   PROCEDURE EscribeMenu;
   BEGIN
      WRITELN('Que deseas hacer? ');
      WRITELN('Existen tres listas, 1, 2 y 3');
      WRITELN('Insercion de elementos al final de una lista--1');
      WRITELN('Insercion de elemento en el lugar i-esimo-----2');
      WRITELN('Retirar el ultimo elemento-------------------3');
      WRITELN('Retirar el i-esimo elemento------------------4');
      WRITELN('Liberar todos los nodos----------------------5');
      WRITELN('Conocer el numero de elementos---------------6');
      WRITELN('Invertir una lista---------------------------7');
      WRITELN('Crear una lista ordenada---------------------8');
      WRITELN('Concatenar dos listas------------------------9');
      WRITELN('Combinar(ordenar) dos listas ordenadas-------10');
      WRITELN('Regresar al menu-----------------------------11');
      WRITELN('Terminar la corrida--------------------------12');
   END;
   {**************************************************************}
   FUNCTION OpcionDeMenu:INTEGER;
   VAR
      opcion:INTEGER;
   BEGIN
      WRITELN; WRITELN;
      WRITE('     Opcion: '); READLN(opcion);
      OpcionDeMenu:=opcion;
   END;
   {**************************************************************}
   PROCEDURE PreguntaCondicion;
   BEGIN
      IF Seleccion in [9,10] THEN
         BEGIN
            WRITE('Que listas ? '); READLN(n1,n2);
         END
      ELSE
         BEGIN
            WRITE('En que lista ? '); READLN(n1);
            IF seleccion IN [1,8] THEN
               BEGIN
                  WRITE('Cuantos elementos ? '); READLN(n3);
                  WRITE('Escribe los elementos: ');
                  FOR i:=1 TO n3 DO  READ(entero[i]);
                  WRITELN;
               END;
            IF seleccion IN[2,4] THEN
               BEGIN
                  WRITE('En que lugar ?   '); READLN(i);
                  IF seleccion=2 THEN
                     BEGIN
                        WRITE('Escribe el entero a insertar:  ');
                        READLN(ent);
```

```
                    END;
                END;
            END;
            CASE n1 OF
              1:   laux:=l1;
              2:   laux:=l2;
              3:   laux:=l3;
           END;
           CASE n2 OF
              1:   laux1:=l1;
              2:   laux1:=l2;
              3:   laux1:=l3;
           END
        END;
      {***********************************************************}
                         {Programa Principal}
BEGIN
   Inicializa;
   EscribeMenu;
   REPEAT
      Seleccion:= OpcionDeMenu;
      CASE Seleccion OF
         1,8:  BEGIN
                  PreguntaCondicion;
                  FOR i:=1 TO n3 DO
                  BEGIN
                     IF Seleccion=1 THEN
                        Insertafinal(laux,entero[i])
                     ELSE
                        Colocaordenando(laux,entero[i]);
                  END;
                  EscribeLista(laux);
               END;
        2,4:   BEGIN
                  PreguntaCondicion;
                  IF seleccion=2 THEN
                     Insertaiesimo(laux,i,ent)
                  ELSE
                  Retiraiesimo(laux,i,ent);
                  EscribeLista(laux);
               END;
          3:   BEGIN
                  PreguntaCondicion;
                  EscribeLista(laux);
                  Retirafinal(laux,ent);
                  EscribeLista(laux);
               END;
          5:   BEGIN PreguntaCondicion; Liberatodos(laux)  END;
          6:   BEGIN PreguntaCondicion; long:=Longitud(laux) END;
          7:   BEGIN
                  PreguntaCondicion;
                  EscribeLista(laux);
                  Invierte(laux);
                  EscribeLista(laux);
               END;
       9,10:   BEGIN
```

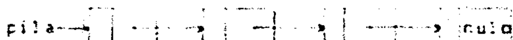
```
                    PreguntaCondicion;
                    IF Seleccion=9 THEN
                       laux2:=Concatena(laux,laux1)
                    ELSE
                       laux3:=Combinaordenando(laux,laux1);
                    EscribeLista(laux);
                    EscribeLista(laux1);
                    IF Seleccion=9 THEN EscribeLista(laux2)
                    ELSE EscribeLista(laux3);
                    CASE n2 OF     (*actualizacion de listas*)
                       1:   l1:=laux1;
                       2:   l2:=laux1;
                       3:   l3:=laux1;
                    END
                 END;
              11: EscribeMenu;
           END;
           CASE n1 OF    (*actualizacion de listas*)
              1:   l1:=laux;
              2:   l2:=laux;
              3:   l3:=laux;
           END;
        UNTIL Seleccion = 12;
     END.
```

```
Que deseas hacer?
Existen tres listas, 1, 2 y 3
Insercion de elementos al final de una lista--1
Insercion de elemento en el lugar i-esimo-----2
Retirar el ultimo elemento-------------------3
Retirar el i-esimo elemento------------------4
Liberar todos los nodos----------------------5
Conocer el numero de elementos---------------6
Invertir una lista---------------------------7
Crear una lista ordenada---------------------8
Concatenar dos listas------------------------9
Combinar(ordenar) dos listas ordenadas-------10
Regresar al menu-----------------------------11
Terminar la corrida--------------------------12


    Opcion: 1
En que lista ? 1
Cuantos elementos ? 4
Escribe los elementos: 2 5 6 9
2--->5--->6--->9--->


    Opcion: 2
En que lista ? 1
En que lugar ?   2
Escribe el entero a insertar:  4
2--->4--->5--->6--->9--->
```

```
    Opcion: 3
En que lista ? 1
2--->4--->5--->6--->9--->
2--->4--->5--->6--->


    Opcion: 4
En que lista ? 1
En que lugar ?   4
2--->4--->5--->


    Opcion: 6
En que lista ? 1
su longitud es  3


    Opcion: 7
En que lista ? 1
2--->4--->5--->
5--->4--->2--->


    Opcion: 8
En que lista ? 3
Cuantos elementos ? 5
Escribe los elementos: 1 8 6 40 20
1--->6--->8--->20--->40--->


    Opcion: 9
Que listas ? 3,1
1--->6--->8--->20--->40--->
5--->4--->2--->
1--->6--->8--->20--->40--->5--->4--->2--->


    Opcion: 5
En que lista ? 1
la lista queda vacia


    Opcion: 12
```

REPRESENTACION DE LA PILA Y COLA CON LISTAS LIGADAS

La pila y cola ligadas son listas, como señalamos anteriormente (casos particulares), no necesariamente secuenciales. Los espacios están unidos por ligas y al hacer inserciones y supresiones se tratan exactamente como pila o como cola.



Pila Ligada

OPERACIONES EN UNA PILA LIGADA

Se agregará un nuevo elemento en el tope y se quitará también por ahí, esto se hace a través del apuntador a la lista ligada. El primer nodo de la lista es el elemento superior o tope de la pila.

Si se tiene un apuntador p a la lista ligada y otro auxiliar t, la operación Agregar(p,x) puede ser implementada por

```
t:=TraeNodo;
espacio[t].inf:=x;
espacio[t].sig:=p;
p:=t;
```

El elemento agregado se saca de la lista de nodos disponibles, llamada espacio. Esta aparece en las declaraciones de listas, páginas 33 y 36, al igual que la variable nulo.

La operación Vacío(p) es sólo para saber si p es igual a nulo.

La operacion Retirar(p) remueve el primer nodo de una lista no vacía y lo manda a la lista de nodos disponibles.

```
IF Vacío(p) THEN
   Error('No hay elementos en la pila')
ELSE
   BEGIN
      t:=p;
      p:=espacio[t].sig;
      x:=espacio[t].inf;
      LiberaNodo(t);
   END;
```

Una de las ventajas de esta implementacion es que todas las pilas que han sido utilizadas por un programa pueden compartir la misma lista de nodos disponibles, tanto para obtener como para regresar nodos. Cada pila tiene la posibilidad de crecer y encoger a cualquier tamaño, considerando la cantidad de espacio disponible. De esta manera no se ha preasignado espacio a una sola pila y ninguna pila utiliza espacio que no necesita.

Una desventaja es el tiempo adicional que se gasta en manejar la lista de nodos disponibles.

Otra estructura de datos que también comparte el mismo conjunto de nodos es la cola.

Cola Ligada

OPERACIONES EN UNA COLA LIGADA

Para manipular las colas necesitamos dos apuntadores: uno al comienzo (frente) y otro al final (fondo).

Supongamos que se tiene un apuntador q a la lista con dos apuntadores, frente y fondo, entonces las operaciones Vacio(q) y Retirar(q) son análogas a Vacio(p) y Retirar(p) con el apuntador frente remplazando a p. Sin embargo, en el caso en que el último elemento se remueve a la variable fondo se le debe asignar el valor nulo, ya que en una cola vacía tanto frente como fondo son nulos. El algoritmo Retirar(q) es por tanto como sigue:

```
IF Vacio(q) THEN
   Error('No hay elementos en la cola')
ELSE
   BEGIN
      t:=q.frente;
      x:=espacio[t].inf;
      q.frente:=espacio[t].sig;
      IF q.frente=nulo THEN
         q.fondo:=nulo;
      LiberaNodo(t);
   END;
```

La operación Agregar(q,x) se puede implementar como sigue:

```
TraeNodo(t);
espacio[t].inf:=x;
espacio[t].sig:=nulo;
WITH q DO
   BEGIN
      IF fondo=nulo THEN
         frente:=t
      ELSE
         espacio[fondo].sig:=t;
      fondo:=t;
   END;
```

```
PROGRAM Banco;
(Ejemplo de aplicacion de listas y colas.
 Simulacion de un banco con 4 cajas, que tendra como resultado el
 tiempo  promedio gastado  por un cliente en el  banco para hacer
 una transaccion, considerando  que el cliente espera en  la cola
 mas corta para  ser atendido, en caso de  que las 4 cajas  esten
 ocupadas.

      datos de entrada- para cada cliente se leeran dos datos:
                             -tiempo de llegada
                             -duracion de la  transaccion a hacer
      datos de salida-  tiempo  promedio que gasta un  cliente en
                        el banco desde  que llega hasta  que ter-
                        mina de ser atendido.

   Los eventos del sistema son  llegada de un cliente o salida de
un cliente.
   A cada  caja se le asocia una estructura de datos de tipo cola
(con implementacion de lista ligada))
CONST
   nulo=0; numnodos=100;
TYPE
   apunt=0..numnodos;
   inftipo=RECORD
             tiempo,duracion: INTEGER;
             tipo: 0..4 (0 si es llegada, 1 salida de la caja 1)
          END;          (2 salida de la caja2, etc.)
   nodotipo=RECORD
              inf: inftipo;
              sig: apunt
           END;
   cola=RECORD
          frente,fondo:apunt;
          num:INTEGER (contiene el numero de clientes)
        END;             (en la cola)
VAR
   espacio:ARRAY[1..numnodos] OF nodotipo;    (nodos disponibles)
   c       :ARRAY[1..4] OF cola;               (las cuatro colas)
   indc    :1..4;       (indice para diferenciar colas)
   evento :apunt;      (apunta a la lista  de eventos)
   tt      :INTEGER;   (tiempo gastado por todos los clientes)
   fin     :BOOLEAN;   (cuando es fin del archivo es TRUE)
   cont    :INTEGER;   (numero de clientes que han pasado al banco)
   tsal    :INTEGER;   (tiempo de salida de un cliente de una cola)
   tll,dur:INTEGER;    (tiempo de llegada de un  cliente y duracion
                        de la transaccion)
   auxinf :inftipo;    (almacena temporalmente la informacion de un
                        nodo)
   ent     :text;      (archivo que  almacena  tiempo  de llegada y
                        duracion de la transaccion de los clientes)
   disponible:apunt; (apunta al primer nodo disponible)

   (*************************************************************)
   PROCEDURE Inicializa;
   (inicializa el espacio disponible, colas, lista, variables y
    archivo de datos de entrada)
```

```
VAR   i:INTEGER;
BEGIN
   FOR i:=1 to numnodos-1 DO (inicializa espacio disponible)
      espacio[i].sig:=i+1;
   espacio[numnodos].sig:=nulo; disponible:=1;
   FOR indc:=1 to 4 DO (inicializa las colas de cada caja)
      WITH c[indc] DO
         BEGIN
            frente:=0;
            fondo:=0;
            num:=0;
         END;
   (inicializa lista)
   evento:=nulo;
   (inicializa variables)
   fin:=FALSE;
   cont:=0;
   tt:=0;
   (inicializa archivo de datos)
   ASSIGN(ent,'DATOS.DAT');
   RESET(ent);
END;

(Se omiten las siguientes rutinas, aparecen anteriormente)
(*****************************************************************)
PROCEDURE Traenodo(VAR p:apunt);
(trae un nodo apuntado por p de la lista de nodos disponibles)
(*****************************************************************)
PROCEDURE Liberanodo(p:apunt);
(agrega un nodo apuntado por p a la lista de disponibles)
(*****************************************************************)
PROCEDURE Insertacabeza(VAR lista:apunt; x:inftipo);
(inserta un nodo con informacion x (evento) a la cabeza de la
 lista de eventos apuntada por lista)
(*****************************************************************)
PROCEDURE Insertadespues(p:apunt; x:inftipo);
(inserta un nodo con informacion x (evento) despues del nodo
 apuntado por p)
(*****************************************************************)
PROCEDURE Retiracabeza(VAR lista:apunt; VAR x:inftipo);
(retira un nodo de la cabeza de la lista de eventos apuntada
por la variable llamada lista)
(*****************************************************************)

PROCEDURE Colocaordenando(VAR lista:apunt; x:inftipo);
(coloca un nodo con informacion x (evento) en orden creciente
 en la lista de eventos apuntada por lista)
VAR  p,q:apunt;
BEGIN
   p:=nulo; q:=lista;
   WHILE (q <> nulo) and (espacio[q].inf.tiempo < x.tiempo) DO
      BEGIN
         p:=q;
         q:=espacio[q].sig;
      END;
   IF p=nulo THEN
```

```
      Insertacabeza(lista,x)
   ELSE
      Insertadespues(p,x);
END;
{************************************************************}
PROCEDURE LeeDatos;
{Lee del archivo los datos de tiempo de llegada y duracion de
 la transaccion  de cada cliente  y lo coloca  en la lista de
 eventos}
BEGIN
   IF NOT EOF(ent) THEN
      WITH auxinf DO
         BEGIN
            READLN(ent,tiempo,duracion);
            tipo:=0;
            Colocaordenando(evento,auxinf);
         END
   ELSE  BEGIN
           fin:=TRUE; CLOSE(ent);
         END;
END;
{************************************************************}
PROCEDURE Agrega(VAR q:cola; x:inftipo);
{agrega un cliente a una cola}
VAR   p:apunt;
BEGIN
   Traenodo(p);
   espacio[p].inf:=x;
   espacio[p].sig:=nulo;
   WITH q DO
      BEGIN
         IF fondo=nulo THEN
            frente:=p
         ELSE
            espacio[fondo].sig:=p;
         fondo:=p; num:=num+1;
      END;
END;
{************************************************************}
PROCEDURE Retira(VAR q:cola; VAR x:inftipo);
{retira un cliente de una cola}
VAR  t:apunt;
BEGIN
   IF q.frente <> nulo THEN
      BEGIN
         t:=q.frente;
         x:=espacio[t].inf;
         q.num:=q.num-1;
         q.frente:=espacio[t].sig;
         IF q.frente=nulo THEN
            q.fondo:=nulo;
         Liberanodo(t);
      END;
END;
{************************************************************}
PROCEDURE Llegada(tll,dur:INTEGER);
```

```
(registra la llegada de un cliente al banco)
VAR t:INTEGER;
    i,j:1..4;
BEGIN
   j:=1;
   t:=c[1].nue;
   FOR i:=2 to 4 DO
      IF c[i].nua < t THEN
         BEGIN
            t:=c[i].nue;
            j:=i;
         END;
   WITH auxinf DO
      BEGIN
         tiempo:=tll;
         duracion:=dur;
         tipo:=j;
      END;
   Agrega(c[j], auxinf);
   (checa si es el unico nodo en la cola, de ser asi la salida
    de ese cliente se pone en la lista de eventos)
   IF c[j].nua=1 THEN
      BEGIN
         auxinf.tiempo:=tll+dur;
         Colocaordenando(evento,auxinf);
      END;
   (si todavia llegan clientes, lee el siguiente y lo coloca
    en la lista de eventos)
   LeeDatos;
END;
(*********************************************************)
PROCEDURE Salida(indc,tsal:INTEGER);
(procedimiento que retira un cliente de una caja)
VAR p:apunt;
BEGIN
   Retira(c[indc],auxinf);
   tt:=tt+(tsal-auxinf.tiempo);
   cont:=cont+1;
   (si hay mas  clientes en la cola, colocar la salida del si-
    guiente cliente en la lista de eventos despues de calcular
    su tiempo de salida)
   IF c[indc].nua > 0 THEN
      BEGIN
         p:=c[indc].frente;
         with auxinf DO
            BEGIN
               tiempo:=tsal+espacio[p].inf.duracion;
               tipo:=indc;
            END;
         Colocaordenando(evento,auxinf);
      END;
END;
(*********************************************************)
PROCEDURE Simula;
(hace simulacion mientras la lista de eventos no esta vacia)
BEGIN
```

```
        WHILE (evento <> nulo) OR (NOT fin) DO
           BEGIN
              Retiracabeza(evento,auxinf);
              (checa si es llegada o salida)
              IF auxinf.tipo=0 THEN  (llegada)
                 BEGIN
                    tll:=auxinf.tiempo;
                    dur:=auxinf.duracion;
                    Llegada(tll,dur);
                 END
              ELSE
                 BEGIN  (salida)
                    indc:=auxinf.tipo;
                    tsal:=auxinf.tiempo;
                    Salida(indc,tsal);
                 END;
           END;
     END;
     (**************************************************************)
     PROCEDURE Escribe;
     BEGIN
        WRITE('El tiempo promedio para ',cont,' clientes es ',
        (tt DIV cont),'.',ROUND(((tt/cont)-TRUNC((tt/cont)))*100));
        WRITELN(' minutos');
    END;
    (**************************************************************)
  BEGIN                        (Programa Principal)
     Inicializa;
     LeeDatos;
     Simula;
     Escribe;
  END.
```

```
A>type DATOS.DAT
     10   10
     15   10
     17   12
     19   20
     20   20
     21    5
     23   10
     25   20
     28   22
     30    2
```

La primera columna del archivo de datos indica el tiempo de llegada de cada cliente. Se considera que a los 10 minutos de abrir el banco comienzan a llegar. En la segunda columna se encuentra el tiempo de duracion (en minutos) de la transaccion de cada cliente.

```
El tiempo promedio para 10 clientes es 18.20 minutos
```

IMPLEMENTACION DE LISTAS CON ESTRUCTURAS DINAMICAS

Una de las ventajas principales que proporciona esta implementación es la de no declarar un conjunto fijo de nodos disponibles a través de un arreglo, para ser utilizados en la ejecución de un programa (mediante las rutinas de Traenodo y Liberanodo), debido a que el número de nodos que se requiere en la mayoría de los casos no se puede predecir en el momento en que se escribe el programa.

En esta implementación no se requiere que el programador esté pendiente de administrar el almacenamiento disponible debido a que el sistema se encarga de la asignación y liberación de nodos.

El método de acceso a un nodo se hace mediante un apuntador, como vimos en la implementación anterior.

Este apuntador es un tipo de datos definido por el programador (TYPE apunt=^t), el cual especifica el tipo de información al que apunta y permite definir variables del mismo tipo (VAR p:apunt). La variable p contiene la dirección de la localidad de memoria que contiene un objeto del tipo t.

Se puede hacer referencia a un nuevo tipo de variable utilizando un apuntador, la construcción p^ es una variable del tipo t.

La mayoría de los compiladores de PASCAL requieren que el tipo de datos al cual es apuntado sea precedido por el símbolo ^.

Para la asignación y liberación de variables dinámicas se utilizan los procedimientos estandar NEW y DISPOSE de PASCAL, respectivamente.

Si p es un apuntador a un objeto del tipo t, NEW(p) crea un objeto del tipo t y asigna su dirección a p. DISPOSE(p) hace que el almacenamiento ocupado por p^ pueda ser reutilizado si es necesario.

Existe un valor especial que cualquier variable tipo apuntador puede tener, el cual es referenciado como nulo (NIL). Este valor de apuntador no hace referencia a una posición de almacenamiento, no apunta a ninguna cosa. Si p apunta a NIL, cualquier referencia a p^ es ilegal.

La implementación de listas ligadas a traves de almacenamiento dinámico puede ser declarada como sigue:

```
TYPE
   apunt=^nodotipo;
   nodotipo=RECORD
               inf:inftipo;
               sig:apunt
            END;

VAR  p:apunt;
```

Las rutinas de Traenodo(p) y Liberanodo(p) de la implementación anterior son reemplazadas por NEW(p) y DISPOSE(p), respectivamente.

A continuación se presentan dos operaciones elementales con implementación dinámica de lista ligada:

```
PROCEDURE Insertacabeza(VAR lista:apunt; x:inftipo);
(Inserta  un  nodo con información x a la lista cuya cabeza  es
apuntada por la variable lista)
VAR
   q:apunt;
BEGIN
   NEW(q); (crea un nodo del tipo especificado y asigna su)
   q^.inf:=x;                              (dirección a q)
   q^.sig:=lista;
   lista:=q;
END;

PROCEDURE Insertadespués(p:apunt; x:inftipo);
(Inserta un nodo con información x, después del nodo p)
VAR
   q:apunt;
BEGIN
   IF p=nil THEN
      Error('la lista está vacía')
   ELSE
      BEGIN
         NEW(q);
         q^.inf:=x;
         q^.sig:=p^.sig;
         p^.sig:=q;
      END;
END;
```

En seguida se presenta otro programa de aplicación de listas.

```
PROGRAM Polinomio;
(Este programa ejecuta la suma de dos polinomios, representados
 en  listas ligadas, con  implementacion  dinamica.  La suma se
 guarda en otra nueva lista)
TYPE
   exponente=0..maxint;
   apunt=^ nodotipo;
   inftipo=RECORD    (guarda el coeficiente y exponente de)
      coef:INTEGER; (cada termino)
      exp:exponente
   END;
   nodotipo=RECORD  (los nodos representan los terminos del)
      inf:inftipo;  (polinomio, ademas  cada nodo  contiene)
      sig:apunt     (un apuntador al siguiente elemento)
   END;
VAR
   a,b,c: apunt;      (apuntadores a los polinomios a sumar y al)
                      (polinomio resultante, respectivamente)

   (Se omiten las siguientes rutinas, aparecen en la definicion
    de operaciones elementales con implementacion dinamica)
   (*******************************************************)
   PROCEDURE Insertacabeza(VAR lista:apunt; x:inftipo);
   (Inserta un nodo con informacion x a la lista cuya cabeza es
    apuntada por la variable llamada lista)
   (*******************************************************)
   PROCEDURE Insertadespues(p:apunt; x:inftipo);
   (Inserta un nodo con inf. x, despues del nodo apuntado por p)
   (*******************************************************)
   PROCEDURE Lee(VAR lista:apunt);
   (Lee un polinomio termino por termino en orden decreciente y
    para concluir lee un termino con coef. y exp. ceros)
   VAR
      q:apunt;  (apuntador para construir la lista que se lee)
      ter:inftipo;
   BEGIN
     IF lista=a THEN
       BEGIN
         WRITE('   Escribe los terminos de los polinomios');
         WRITELN(' en orden');
         WRITELN('decreciente y como ultimo termino escribe 0 0');
         WRITELN;
       END;
     WRITE('          coef.  y  exp.    '); READLN(ter.coef,ter.exp);
     WHILE ter.coef <> 0 DO
        BEGIN
           IF lista=NIL THEN
              BEGIN
                 Insertacabeza(lista, ter);
                 q:=lista;
              END
           ELSE
              BEGIN
                 Insertadespues(q, ter);
                 q:=q^.sig;
              END;
```

```
            WRITE('         coef.  y  exp.   ');
            READLN(ter.coef, ter.exp);
         END;
   END;
   (***********************************************************)
   Function Suma(lista1,lista2:apunt):apunt;
   (Esta funcion ejecuta la suma de los polinomios representados
    en  las listas  apuntadas por  lista1 y lista2 regresando un
    apuntador a la lista que guarda la suma)
   VAR
      p,q:apunt;    (apuntadores para recorrer las listas a sumar)
      r:apunt;      (apuntador para construir la lista de la suma)
      aux,t:apunt; (apuntadores auxiliares)
      s:INTEGER;    (guarda la suma de coeficientes)
      ter:inftipo; (variable auxiliar para guardar un termino)
   BEGIN
      p:=lista1; q:=lista2;
      New(t);        (nodo auxiliar que al final se retira)
      t^.sig:=NIL;
      suma:=t; r:=t;
      WHILE (p <> NIL) and (q <> NIL) DO
         BEGIN
            IF p^.inf.exp > q^.inf.exp THEN
               BEGIN
                  ter:=p^.inf;
                  p:=p^.sig;
               END
            ELSE
               IF p^.inf.exp < q^.inf.exp THEN
                  BEGIN
                     ter:=q^.inf;
                     q:=q^.sig;
                  END
               ELSE
                  BEGIN
                     s:=p^.inf.coef + q^.inf.coef;
                     IF s <> 0 THEN
                        BEGIN
                           ter.coef:=s;
                           ter.exp:=p^.inf.exp;
                        END;
                     p:=p^.sig; q:=q^.sig;
                  END;
            IF ter.exp <> r^.inf.exp THEN
            BEGIN
               Insertadespues(r,ter);
               r:=r^.sig;
            END;
         END;
      IF p <> NIL THEN
         aux:=p
      ELSE
         aux:=q;
      WHILE aux <> NIL DO
         BEGIN
            Insertadespues(r, aux^.inf);
```

```
               aux:=aux^.sig;
               r:=r^.sig;
            END;
         Suma:=t^.sig;
         Dispose(t);  {se libera el nodo auxiliar apuntado por t}
      END;
      {***********************************************************}
      PROCEDURE Escribe(lista:apunt);
      {Escribe el polinomio representado en una lista}
      BEGIN
         IF lista=c THEN  WRITELN(' ':8,'El polinomio suma es:  ')
         ELSE WRITE(' ':31);
         REPEAT
            IF lista^.inf.coef < 0 THEN
               WRITE('-');
            IF lista^.inf.coef <> 1 THEN
               WRITE(abs(lista^.inf.coef));
            IF lista^.inf.exp > 0 THEN
               IF lista^.inf.exp <> 1 THEN
                  WRITE('x^',lista^.inf.exp)
               ELSE  WRITE('x');
            lista:=lista^.sig;
            IF lista <> NIL THEN
               IF lista^.inf.coef > 0 THEN
                  WRITE('+');
         UNTIL lista=NIL;
         WRITELN; WRITELN;
      END;
      {***********************************************************}
                           {Programa Principal}
   BEGIN
      a:=NIL; b:=NIL; c:=NIL;
      Lee(a); Escribe(a);
      Lee(b); Escribe(b);
      c:=Suma(a,b); Escribe(c);
   END.
```

```
Escribe los terminos de los polinomios en orden
decreciente y como ultimo termino escribe 0 0

      coef.  y  exp.     6  7
      coef.  y  exp.     1  6
      coef.  y  exp.    -2  3
      coef.  y  exp.     0  0
                            6x^7+x^6-2x^3

      coef.  y  exp.     4  9
      coef.  y  exp.    60  6
      coef.  y  exp.     8  3
      coef.  y  exp.     9  0
      coef.  y  exp.     0  0
                            4x^9+60x^6+8x^3+9

      El polinomio suma es:  4x^9+6x^7+61x^6+6x^3+9
```

Otras estructuras de listas:

LISTAS CIRCULARES

Una lista circular se obtiene al hacer un pequeño cambio en la estructura de una lista lineal. Este cambio consiste en que el campo siguiente (sig) del último nodo contiene un apuntador que va al primer nodo, en lugar de ser el apuntador nulo. En estas listas una de las características es que desde cualquier nodo es posible alcanzar o llegar a otro nodo cualquiera de la lista.

LISTAS CON NODOS DE ENCABEZAMIENTO

Nodo de encabezamiento es un nodo extra que se encuentra al frente o inicio de una lista cuando así se desea. Este nodo no representa un elemento de la lista; la porción de información de este nodo es comúnmente utilizada para mantener la información global con respecto a toda la lista, como puede ser el número de nodos en la lista.

LISTAS MULTIENCADENADAS

Una lista multiencadenada o multiligada es una estructura en donde los nodos aparecen en más de una lista y contienen más de un apuntador.

Podemos ver en la página 96 una figura que ilustra la representación multiencadenada de una matriz dispersa.

Estas listas multiencadenadas proporcionan aún más ventajas (aunque requieren más espacio para representarlas) que las listas ligadas, como es la de poder eliminar un nodo dando solamente un apuntador a este nodo.

Las listas multiencadenadas pueden ser circulares o lineales y contener o no nodos de encabezamiento.

Un caso particular de las listas multiligadas son las listas doblemente ligadas, donde cada nodo contiene dos apuntadores, uno para su predecesor y el otro para su sucesor.

A continuación presentamos un ejemplo de aplicación de listas multiencadenadas, con implementación dinámica.

El ejemplo de aplicación es calcular la suma de dos matrices dispersas.

Mantendremos en la memoria una estructura que contiene solamente aquellos elementos que son diferentes a cero, encadenando uno a otro en una forma que describa su posición en la matriz; como se muestra en la figura de la página 96.

Con lo que respecta a las operaciones básicas de listas multiencadenadas, se dan algunas en el programa siguiente.

```
PROGRAM Suma;
( Ejecuta la suma de dos matrices dispersas, A y B, guardando la
 suma en la matriz B)
TYPE
   apunt=^nodotipo;
   nodotipo=RECORD     (estructura de cada nodo)
               ren: INTEGER;    (numero de renglon)
               col: INTEGER;    (numero de columna)
               valor: INTEGER; (valor de una entrada de la matriz)
               sigren: apunt;  (elemento sig. en el mismo renglon)
               sigcol: apunt    (elemento sig. en la misma columna)
            END;
VAR
   A,B: apunt;    (matrices a sumar)
   m,n: INTEGER; (limites de la matriz, renglon y columna)

   (*****************************************************************)
   PROCEDURE InsertaDespues(p:apunt; r,c:INTEGER);
   (inserta un nodo con informacion r y c despues del nodo apunta-
    do por p)
   VAR
      q:apunt;
   BEGIN
      IF p=NIL THEN
         WRITELN('Insercion de un vacio')
      ELSE
         BEGIN
            NEW(q);
            q^.ren:=r;
            q^.col:=c;
            IF c=0 THEN
               BEGIN
                  q^.sigren:=p^.sigren;
                  q^.sigcol:=q;
                  p^.sigren:=q;
               END
            ELSE
               BEGIN
                  q^.sigcol:=p^.sigcol;
                  q^.sigren:=q;
                  p^.sigcol:=q;
               END;
         END;
   END;
   (*****************************************************************)
   FUNCTION CreaCabezas(m,n:INTEGER):apunt;
   (esta funcion  crea el renglon y columna de  encabezamiento,
    regresando un apuntador al nodo que los une, siendo este el
    nodo cabeza de la matriz)
   VAR
      cabeza, ultimo: apunt;
      r,c: INTEGER;
   BEGIN
      NEW(cabeza);
      WITH cabeza^ DO
         BEGIN
```

```
            ren:=0;  col:=0;
            sigren:=cabeza;  sigcol:=cabeza;
         END;
      ultimo:=cabeza;
      FOR r:=1 TO  m  DO
         BEGIN
            (* crea la columna de encabezamiento para renglones *)
            InsertaDespues(ultimo,r,0);
            ultimo:=ultimo^.sigren;
         END;
      ultimo:=cabeza;
      FOR c:=1 TO  n  DO
         BEGIN
            (* crea el renglon de encabezamiento para  columnas *)
            InsertaDespues(ultimo,0,c);
            ultimo:=ultimo^.sigcol;
         END;
      CreaCabezas:=cabeza;
   END;
   (*****************************************************************)
   FUNCTION BuscaRen(r,c:INTEGER; matriz:apunt):apunt;
   (localiza el renglon r, recorre este si hay nodos hasta llegar
   a la columna  mas grande y menor que c, se posiciona en este
   elemento regresando un apuntador a el)
   VAR
      numcol:INTEGER;
      aptren:apunt;
   BEGIN
      aptren:=matriz^.sigren;
      WHILE aptren^.ren < r DO
         aptren:=aptren^.sigren;
      numcol:=aptren^.sigcol^.col;
      WHILE (numcol < c) AND (numcol >0)  DO
         BEGIN
            aptren:=aptren^.sigcol;
            numcol:=aptren^.sigcol^.col;
         END;
      IF numcol=c THEN
         BuscaRen:=NIL
      ELSE
         BuscaRen:=aptren;
   END;
   (*****************************************************************)
   FUNCTION BuscaCol(r,c:INTEGER; matriz:apunt):apunt;
   (localiza la columna c, recorre esta si hay nodos hasta llegar
    al renglon  mas grande y menor  que r, se posiciona en este
    elemento regresando un apuntador a el)
   VAR
      numren: INTEGER;
      aptcol:apunt;
   BEGIN
      aptcol:=matriz^.sigcol;
      WHILE aptcol^.col < c DO
         aptcol:=aptcol^.sigcol;
         numren:=aptcol^.sigren^.ren;
         WHILE (numren < r) AND (numren > 0) DO
```

```
              BEGIN
                 aptcol:=aptcol^.sigren;
                 numren:=aptcol^.sigren^.ren;
              END;
           IF numren=r THEN
              BuscaCol:=NIL
           ELSE
              BuscaCol:=aptcol;
END;
{**************************************************************}
PROCEDURE InsertaNodo(r,c,v:INTEGER; matriz:apunt);
{inserta un nodo con valor v en el renglon r y columna c de
 la matriz}
VAR
   aptren,aptcol,aptnodo:apunt;
BEGIN
   aptren:=BuscaRen(r,c,matriz);
   aptcol:=BuscaCol(r,c,matriz);
   IF (aptren=NIL) AND (aptcol=NIL) THEN
      WRITELN('ERROR: ya existe nodo')
   ELSE
      IF (aptren=NIL) OR (aptcol=NIL) THEN
         WRITELN('ERROR: inconsistencia')
      ELSE
         IF aptren=aptcol THEN
            WRITELN('RROR: matriz mal construida')
         ELSE
            BEGIN
               NEW(aptnodo);
               WITH aptnodo^ DO
                  BEGIN
                     ren:=r;  col:=c; valor:=v;
                     sigren:=aptcol^.sigren;
                     sigcol:=aptren^.sigcol;
                  END;
               aptcol^.sigren:=aptnodo;
               aptren^.sigcol:=aptnodo;
            END;
END;
{**************************************************************}
PROCEDURE LeeMatriz(VAR matriz:apunt);
VAR
   r,c,v:INTEGER;  {renglon, columna y valor respectivamente}
BEGIN
  IF matriz=A THEN
  BEGIN
   WRITE('Escribe las dimensiones de las matrices a sumar: ');
   READLN(m,n);
   WRITELN('Escribe las matrices por renglones'); WRITELN;
  END;
   matriz:=CreaCabezas(m,n);
   FOR r:=1 TO m DO
      BEGIN  WRITE(' ':15);
         FOR c:=1 TO n DO
            BEGIN
               READ(v);
```

```
              IF v <> 0 THEN
                 InsertaNodo(r,c,v,matriz);
           END;
        READLN;
     END; WRITELN;
END;
{**************************************************************}
PROCEDURE Suma(A:apunt; VAR B:apunt);
(se suman las matrices A y B, guardando la suma en B)
VAR
   i:INTEGER;
   r1,r2,aux1,aux2:apunt;
  {************************************************************}
   PROCEDURE Sumar;
   BEGIN
      aux2^.valor:=aux1^.valor + aux2^.valor;
      r1:=aux1;  aux1:=aux1^.sigcol;
      r2:=aux2;  aux2:=aux2^.sigcol;
   END;
   {***********************************************************}
   PROCEDURE Recorrer;
   BEGIN
      WHILE (aux2^.col < aux1^.col) AND (aux2^.col <> 0) DO
         BEGIN
            r2:=aux2;  aux2:=aux2^.sigcol;
         END;
   END;
   {***********************************************************}
   PROCEDURE Insertar;
   BEGIN
      InsertaNodo(aux1^.ren, aux1^.col, aux1^.valor, B);
      aux1:=aux1^.sigcol;
   END;

BEGIN                        (* empieza suma *)
  FOR i:=1 TO m DO
    BEGIN   (* el control de la suma se lleva por renglones *)
      r1:=A;  r2:=B;
      WHILE (r1^.sigren^.ren <= i)AND(r1^.sigren^.ren <> 0) DO
         BEGIN
            r1:=r1^.sigren;  r2:=r2^.sigren;
         END;
      aux1:=r1^.sigcol;  aux2:=r2^.sigcol;
      WHILE (aux1^.col <> 0) AND (aux2^.col <> 0) DO
         BEGIN
            IF aux1^.col = aux2^.col THEN Sumar
            ELSE
               IF aux1^.col < aux2^.col THEN Insertar
               ELSE
                  IF aux1^.col > aux2^.col THEN Recorrer;
         END;
      IF aux2^.col = 0  THEN
         WHILE aux1^.col <> 0 DO  Insertar;
    END;
END;
```

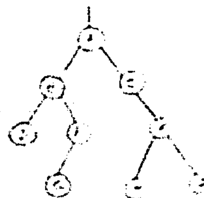
```
(****************************************************************)
PROCEDURE Escribe(matriz:apunt);
VAR
   r,c:INTEGER;
   p,q:apunt;
BEGIN
   WRITELN(' ':15,'Matriz suma');
   p:=matriz;
   FOR r:=1 TO m DO
       BEGIN  WRITE(' ':15);
          p:=p^.sigren;  q:=p^.sigcol;
          FOR c:=1 TO n DO
              IF q^.col=c THEN
                 BEGIN
                    WRITE(q^.valor,'   ');
                    q:=q^.sigcol;
                 END
              ELSE
                 WRITE('0   ');
          WRITELN;
       END;
END;
(****************************************************************)
                              { Programa Principal }
BEGIN
   LeeMatriz(A);
   LeeMatriz(B);
   Suma(A,B);
   Escribe(B);
END.
```

```
Escribe las dimensiones de las matrices a sumar: 4 4
Escribe las matrices por renglones

                  0  0  2  0
                  0  0  0  0
                  0  0  0  3
                  1  0  0  4

                  1  0  0  0
                  0  0  0  0
                  0  0  0  3
                  1  0  0  1

                  Matriz suma
                  1  0  2  0
                  0  0  0  0
                  0  0  0  6
                  2  0  0  5
```

ARBOLES BINARIOS

DEFINICION

Arbol binario es un conjunto finito de elementos que puede estar vacío o contener un elemento distinguido llamado raíz; el resto se particiona en dos subconjuntos ajenos, cada uno de los cuales es un árbol binario. Estos dos subconjuntos reciben el nombre de subárbol izquierdo y subárbol derecho del árbol original. Cada elemento de un árbol binario se denomina un nodo del árbol.



Un Arbol Binario

Si n1 es la raíz de un árbol binario y n2 es la raíz del subárbol izquierdo o derecho de ese mismo árbol entonces se dice que n1 es el padre de n2 y n2 es el hijo izquierdo o derecho de n1. Se denomina hoja a un nodo que no tiene hijos.

OPERACIONES BASICAS

Consideramos que cada nodo tiene tres campos elementales. Uno de estos contiene la información del nodo y los otros dos los apuntadores a las raíces de los subárboles izquierdo y derecho.

De aquí que al construir un árbol binario se utilicen las operaciones ConstruyeArbol, ConjuntoIzq y ConjuntoDer. En este caso ConstruyeArbol(x) crea un árbol binario de un solo nodo con un campo de información x, sus hijos izquierdo y derecho apuntan a NIL y retorna un apuntador a este nodo. ConjuntoIzq(p,x) acepta un apuntador p de un nodo de un árbol binario y crea un hijo izquierdo del nodo apuntado por p con un campo de información x. ConjuntoDer(p,x) es análoga a ConjuntoIzq(p,x), sin embargo crea un hijo derecho del nodo apuntado por p.

Así como agregar nodos en un árbol, nos interesa quitarlos. Esta operación la veremos más adelante, en los temas de búsqueda y ordenamiento.

Después de construir un árbol binario otra operación básica muy útil es la de recorrer el árbol. Esto es visitar cada uno de los nodos del árbol para ejecutar una determinada operación.

Definiremos tres métodos de manera recursiva para hacer este recorrido, de tal manera que recorrer un árbol binario comprende la visita de la raíz y el recorrido de los subárboles izquierdo y derecho. La diferencia entre los métodos es el orden en que se realizan estas tres operaciones.

Para recorrer un árbol binario no vacío en **preorden:**

- Visitar la raíz
- Recorrer el subárbol de la izquierda en preorden

- Recorrer el subarbol de la derecha en preorden

Para recorrer un árbol binario no vacío en inorden:

- Recorrer el subarbol de la izquierda en inorden
- Visitar la raíz
- Recorrer el subarbol de la derecha en inorden

Para recorrer un árbol binario no vacío en posorden:

- Recorrer el subarbol de la izquierda en posorden
- Recorrer el subarbol de la derecha en posorden
- Visitar la raíz

IMPLEMENTACION DE ARBOLES BINARIOS EN PASCAL

La implementación de arboles binarios se lleva a cabo de igual forma que la implementación de listas, esto es a través de un arreglo o de estructuras dinámicas.

IMPLEMENTACION DE ARBOLES BINARIOS CON ESTRUCTURAS DINAMICAS

Esta implementación puede ser declarada como sigue:

```
TYPE
   apunt=^nodotipo;
   nodotipo=RECORD
               inf:inftipo;
               hijoizq:apunt;
               hijoder:apunt
            END;
VAR
   p:apunt;
```

Las operaciones básicas son:

```
FUNCTION ConstruyeArbol(x:inftipo):apunt;
{crea un arbol de un  nodo con información x, sus hijos apuntan
 a NIL y retorna un apuntador a este nodo}
VAR
   p:apunt;
BEGIN
   NEW(p);
   p^.inf:=x;
   p^.hijoizq:=NIL;
   p^.hijoder:=NIL;
   ConstruyeArbol:=p;
END;

PROCEDURE ConjuntoIzq(p:apunt; x:inftipo);
{crea un hijo izquierdo de un nodo}
VAR
   q:apunt;
BEGIN
   IF p^.hijoizq <> NIL THEN
      Error('operación ConjuntoIzq ilegal')
   ELSE
      BEGIN
         q:=ConstruyeArbol(x);
         p^.hijoizq:=q;
      END;
END;
```

```
PROCEDURE ConjuntoDer(p:apunt; x:inftipo);
{crea un hijo derecho de un nodo}
VAR
   q:apunt;
BEGIN
   IF p^.hijoder <> NIL THEN
      Error('operación ConjuntoDer ilegal').
   ELSE
      BEGIN
         q:=ConstruyeArbol(x);
         p^.hijoder:=q;
      END;
END;
```

Los siguientes procedimientos recursivos recorren un arbol; reciben de parametro un apuntador (a) al arbol con el cual se opera. El parametro implícito (op) indica la operación a realizar en cada nodo.

```
PROCEDURE Preorden(a:apunt);
BEGIN
   IF a <> NIL THEN
      BEGIN
         op(a);
         Preorden(a^.hijoizq);
         Preorden(a^.hijoder);
      END;
END;

PROCEDURE Inorden(a:apunt);
BEGIN
   IF a <> NIL THEN
      BEGIN
         Inorden(a^.hijoizq);
         op(a);
         Inorden(a^.hijoder);
      END;
END;

PROCEDURE Posorden(a:apunt);
BEGIN
   IF a <> NIL THEN
      BEGIN
         Posorden(a^.hijoizq);
         Posorden(a^.hijoder);
         op(a);
      END;
END;
```

Una aplicación de los Arboles Binarios.

Dada una lista de números en un archivo, se desea ordenar éstos en forma ascendente.

El problema se resuelve con un arbol binario, en el que los números menores que la raíz se encuentran en el subarbol izquierdo y los mayores o iguales en el subarbol derecho. Así, al recorrer (imprimiendo la información de sus nodos) este árbol en inorden, obtenemos los números ordenados en forma ascendente.

ALGORITMO

El primer numero es leído y colocado en un nodo que se establece como la raíz del arbol binario, con subárboles izquierdo y derecho vacíos. Cada numero sucesivo en la lista es comparado con el numero de la raíz. Si es menor, el proceso se repite con el subarbol de la izquierda y si es mayor o igual el proceso se repite con el subarbol de la derecha hasta que se llega a algún subárbol vacío. En este caso el número se coloca en un nuevo nodo en esa posición del arbol.

```
PROGRAM  Ordena;
{Este programa  ordena una lista de numeros, a traves de un arbol
 binario}
TYPE
   inftipo=INTEGER;  {es el tipo de informacion de cada nodo}
   apunt=^nodotipo;  {apuntador a los nodos del arbol}
   nodotipo=RECORD   {tipo de nodos del arbol}
               inf:inftipo;
               hijoizq:apunt;
               hijoder:apunt
            END;
VAR
   arbol:apunt;       {apuntador al arbol principal}

   {*************************************************************}
   FUNCTION FormaArbol:apunt;
   {crea un arbol binario que representa los numeros y regresa un
    apuntador a este arbol}
   VAR
      num:INTEGER;     {guarda los enteros a leer del archivo}
      ent:text;        {archivo de los numeros a ordenar}
      p,q,r:apunt;     {apuntadores auxiliares}
      {**********************************************************}
      FUNCTION construyeArbol(x:inftipo):apunt;
      {crea un arbol binario de un solo nodo}
      VAR
         p:apunt;
      BEGIN
         NEW(p);
         p^.inf:=x;
         p^.hijoizq:=NIL;
         p^.hijoder:=NIL;
         ConstruyeArbol:=p;
      END;
      {**********************************************************}
      PROCEDURE ConjuntoIzq(p:apunt; x:inftipo);
      {crea el hijo izquierdo del nodo p, con informacion x}
      VAR
         q:apunt;
      BEGIN
         IF p^.hijoizq <> NIL THEN
            WRITELN('operacion ConjuntoIzq ilegal')
         ELSE
            BEGIN
               q:=ConstruyeArbol(x);
               p^.hijoizq:=q;
            END;
      END;
      {**********************************************************}
      PROCEDURE ConjuntoDer(p:apunt; x:inftipo);
      {crea el hijo derecho del nodo p, con informacion x}
      VAR
         q:apunt;
      BEGIN
         IF p^.hijoder <> NIL THEN
            WRITELN('operacion ConjuntoDer ilegal')
```

```
        ELSE
           BEGIN
              q:=ConstruyeArbol(x);
              p^.hijoder:=q;
           END;
    END;
{**********************************************************}
                    (funcion FormaArbol)
BEGIN
    ASSIGN(ent,'DATOS.DAT'); RESET(ent);
    READ(ent,num);               (el primer numero es leido)
    r:=ConstruyeArbol(num);  (se construye su arbol)
    FormaArbol:=r;   (retorna un apuntador al arbol principal)
    WHILE NOT EOF(ent) DO
        BEGIN
           READ(ent,num);
           (cada numero  sucesivo de la lista es comparado con
            la raiz, si es menor  el proceso se repite  con el
            subarbol isquierdo, si no con el subarbol derecho)
           q:=r;
           WHILE q <> NIL DO
              BEGIN
                 p:=q;
                 IF num < p^.inf THEN
                    q:=p^.hijoizq
                 ELSE
                    q:=p^.hijoder;
              END;
             (este proceso termina cuando se llega a un subarbol
              vacio, en  este  caso el numero se coloca en un
              nuevo nodo en esa posicion en el subarbol)
           IF num < p^.inf THEN
              ConjuntoIzq(p,num)
           ELSE
              ConjuntoDer(p,num);
        END;
END;
{**********************************************************}
PROCEDURE Recorre(a:apunt);

    PROCEDURE Inorden(a:apunt);
    (recorre el arbol en Inorden)
    BEGIN
        IF a <> NIL THEN
           BEGIN
              Inorden(a^.hijoizq);
              WRITE(a^.inf,'  ');  (operacion a realizar)
              Inorden(a^.hijoder);
           END;
    END;
                 (* inicia Recorre *)
BEGIN
    Inorden(a);(recorre el arbol imprimiendo la informacion de)
    WRITELN;   (sus nodos)
    WRITELN('      Numeros ordenados en  forma ascendente');
END;
```

```
   {*************************************************************}
                     { Programa Principal }
BEGIN
   arbol:=FormaArbol;
   Recorre(arbol);
END.
```

```
A>type DATOS.DAT
14  15  100  9  7  18  3  8  16  4  20  17  49  14  65


3  4  7  8  9  14  14  15  16  17  18  20  49  65  100
     Numeros ordenados en  forma ascendente
```

Otra aplicación de los Arboles Binarios.

Convertir una expresión aritmética infija en prefija, sufija e infija misma y evaluar dicha expresión. En este ejemplo los operandos de la expresión aritmética son de un solo dígito.

Para obtener las notaciones equivalentes, se construye un árbol, el cual representa la expresión, y se recorre este árbol en los tres métodos distintos. La evaluación de la expresión también hace uso del árbol binario.

Este ejercicio ya se ha resuelto con la utilización de pilas (ver tema de pilas). El procedimiento que se lleva a cabo es bastante similar, con la diferencia de que aquí no existe cuerda de salida, la salida es un apuntador al árbol que se construye.

ALGORITMO
Se analiza la cuerda de entrada.
Si se encuentra con un operando
    Se construye su árbol y el apuntador a este se coloca a la pila de árboles.
Si se encuentra con un operador, existen diferentes casos:
    Si la pila de operadores está vacía
        Este operador se inserta en la pila
    Si la pila no está vacía
        Se analiza la prioridad de este operador con respecto al operador que se encuentra en el tope de la pila
            Si el operador tope de la pila es de mayor prioridad
                Se saca y se construye su árbol asignándole como hijos los dos últimos árboles, los cuales son retirados de la pila de árboles. Este árbol resultante se agrega a la pila de árboles
                Se agrega el operador presente a su respectiva pila.
            Si no es de mayor prioridad
                Nada se saca de la pila y se agrega el operador presente en la pila
Este proceso se repite hasta terminar de analizar toda la cuerda. Si llegan a quedar operadores en la pila se sacan (de uno en uno) y se construye su árbol asignándoles como hijos a los dos últimos árboles de la pila de árboles. Este árbol que se construye se coloca a la pila de árboles, así hasta que la pila de operadores quede vacía. El árbol resultante es el que queda en la pila de árboles.

Al tener ya formado el árbol se recorre en preorden y posorden para obtener la expresión aritmética en notaciones prefija y sufija, respectivamente. Para obtener la expresión infija cuya forma requiere de paréntesis no basta recorrer el árbol en Infija, ya que el árbol no contiene paréntesis, pues el orden de las operaciones está indicado por la estructura del árbol. Por tanto hay que escribir los paréntesis.

En lo que respecta a la evaluación, ésta se lleva a cabo en la función EvalúaArbol, la cual recibe como parámetro un apuntador al árbol formado anteriormente y retorna el valor de la evaluación. Utiliza la función Opera, la que ejecuta las operaciones, regresando un valor. La evaluación se realiza de manera recursiva.

```
PROGRAM Exprarit;
( Dada una expresion aritmetica en notacion infija, construir el
 arbol que la representa y recorrer dicho arbol en inorden, pre-
 orden y  posorden,  para asi obtener  la expresion en  notacion
 infija, prefija y sufija,  respectivamente.  El arbol se evalua
 para obtener el valor de la expresion aritmetica)
CONST
   maxexpr=80;  (maxima longitud de la expresion aritmetica)
TYPE
   expr=PACKED ARRAY[1..maxexpr] OF CHAR;
   infitipo=CHAR;
   apunt=^nodotipo;
   nodotipo=RECORD
                inf:infitipo;
                hijoizq:apunt;
                hijoder:apunt
            END;
VAR
   arbol:apunt;  (apuntador al arbol resultante)
   i:INTEGER;    (variable auxiliar)
   ce:expr;      (cuerda de entrada)

   (**********************************************************)
   PROCEDURE IniciayLee(VAR ce:expr);
   (inicializa la cuerda de entrada y la lee)
   BEGIN
      FOR i:=1 TO maxexpr DO  ce[i]:=' ';
      WRITELN('Escribe la expresion infija a tratar');
      READLN(ce);
   END;

   (Rutina que asigna la prioridad de los operadores.
    Se omite, aprece en la pagina 16)
   (**********************************************************)
   FUNCTION Prcd(opdr1,opdr2:CHAR):BOOLEAN;
   (asigna la prioridad de los operadores. Es verdadero cuando la
    prioridad de opdr1 es mayor que opdr2)
   (**********************************************************)

   FUNCTION FormaArbol(ce:expr):apunt;
   (forma el arbol que representa la expresion aritmetica)
   CONST
      maxpila=80;    (maxima longitud de las pilas)
   TYPE
      pila1tipo=CHAR;
      pila1=RECORD
                elem:ARRAY[1..maxpila]OF pila1tipo;
                tope:0..maxpila
            END;
      pila2tipo=apunt;
      pila2=RECORD
                elem:ARRAY[1..maxpila] OF pila2tipo;
                tope:0..maxpila
            END;
   VAR
      apunt1,apunt2:apunt; (apuntadores a  hijos izq. y der.)
```

```
opdr,simb:CHAR;         (var. auxiliar y simbolo presente)
arbolp:pila2;           (pila de apuntadores a arboles)
i:1..maxexpr;           (indica posicion de ce)
opdrp:pila1;            (pila de operadores)
arbolaux:apunt;         (apuntador auxiliar)

(Rutinas para la manipulacion de pilas)
(Estas rutinas son las ya tratadas en el tema de pilas,
 por esta razon las omitiremos)
(*****************************************************)
PROCEDURE Agregar1(VAR p:pila1; x:pila1tipo);
(agrega un elemento a la pila de operadores)
(*****************************************************)
FUNCTION Vacio1(p:pila1):BOOLEAN;
(prueba si la pila de operadores esta vacia)
(*****************************************************)
FUNCTION Retirar1(VAR p:pila1):pila1tipo;
(retira el ultimo elemento de la pila de operadores)
(*****************************************************)
PROCEDURE Agregar2(VAR p:pila2; x:pila2tipo);
(agrega un elemento a la pila de arboles)
(*****************************************************)
FUNCTION Vacio2(p:pila2):BOOLEAN;
(prueba si la pila de arboles esta vacia)
(*****************************************************)
FUNCTION Retirar2(VAR p:pila2):pila2tipo;
(retira el ultimo elemento de la pila de arboles)

(Rutinas para la formacion del arbol)
(tambien las omitimos, aparecen en el programa Ordena)
(*****************************************************)
FUNCTION ConstruyeArbol(x:inftipo):apunt;
(crea un arbol binario de un nodo, sus hijos apuntan a NIL)
(*****************************************************)
PROCEDURE ConjuntoIzq(p:apunt; x:inftipo);
(crea un nuevo hijo izquierdo)
(*****************************************************)
PROCEDURE ConjuntoDer(p:apunt; x:inftipo);
(crea un nuevo hijo derecho)
(*****************************************************)

PROCEDURE RetyAgr;
(esta rutina retira el ultimo operador de la pila; con esta
 informacion  construye un arbolito  de un solo nodo, donde
 sus hijos apuntan a los dos ultimos arboles que se retiran
 de la pila correspondiente)
VAR  arbolito:apunt;  (apuntador auxiliar)
BEGIN
   opdr:=Retirar1(opdrp);
   arbolito:=ConstruyeArbol(opdr);
   apunt2:=Retirar2(arbolp);
   apunt1:=Retirar2(arbolp);
   arbolito^.hijoder:=apunt2;
   arbolito^.hijoizq:=apunt1;
   Agregar2(arbolp, arbolito);
END;
```

```
(********************************************************************)
               (procedimiento FormaArbol)
BEGIN
   opdrp.tope:=0;  (se inicializa tope pila de operadores)
   arbolp.tope:=0; (se inicializa tope pila de arboles)
   i:=1;           (indica la posicion de ce)
   simb:=ce[i];    (indica el simbolo presente)
   (se analizan los simbolos de entrada)
   WHILE simb <> ' ' DO
      BEGIN
         IF simb IN ['a'..'z','0'..'9'] THEN
            BEGIN
               arbolaux:=ConstruyeArbol(simb);
               Agregar2(arbolp, arbolaux);
            END
         ELSE
            BEGIN
               WHILE((NOT Vacio1(opdrp)) AND
                  (Prcd(opdrp.elem[opdrp.tope],simb))) DO
                     RetyAgr;
               IF Vacio1(opdrp) OR (simb <> ')') THEN
                  Agregar1(opdrp,simb)
               ELSE
                  opdr:=Retirar1(opdrp);
            END;
         IF i < maxexpr THEN
            BEGIN
               i:=i+1; simb:=ce[i];
            END
         ELSE
            simb:=' ';
      END;
   WHILE (NOT Vacio1(opdrp)) DO  RetyAgr;
   FormaArbol:=Retirar2(arbolp);
END;
(********************************************************************)
PROCEDURE Recorre(a:apunt);
(rutina para recorrer un arbol)
VAR b:apunt;
   (****************************************************************)
   PROCEDURE Inorden(a,b:apunt);
   VAR aux:BOOLEAN;

      FUNCTION Parentesis:BOOLEAN;
      BEGIN
        Parentesis:=FALSE;
        IF (a^.inf IN ['+','-','*','/','$'] ) AND
           Prcd(b^.inf, a^.inf) THEN
        IF ((b^.inf <> a^.inf) AND ((b^.inf) IN ['*','/','$']))
           OR ((b^.inf='-')AND(a^.inf IN ['+','-']))AND
           (b^.hijoder=a) THEN
           Parentesis:=TRUE;
      END;
   BEGIN         (* inicia Inorden *)
      IF a <> NIL THEN
         BEGIN
```

```
                aux:=Parentesis; IF aux THEN WRITE('(');
                Inorden(a^.hijoizq, a);
                WRITE(a^.inf);
                Inorden(a^.hijoder, a);
                aux:=Parentesis; IF aux THEN WRITE(')');
             END;
       END;
       {**********************************************************}
       PROCEDURE Preorden(a:apunt);
       BEGIN
          IF a <> NIL THEN
             BEGIN
                WRITE(a^.inf);
                Preorden(a^.hijoizq);
                Preorden(a^.hijoder);
             END;
       END;
       {**********************************************************}
       PROCEDURE Posorden(a:apunt);
       BEGIN
          IF a <> NIL THEN
             BEGIN
                Posorden(a^.hijoizq);
                Posorden(a^.hijoder);
                WRITE(a^.inf);
             END;
       END;
                   (* inicia procedimiento Recorre *)
    BEGIN
       WRITE('Recorrido en Inorden:   '); NEW(b); b^.inf:='*';
       Inorden(a,b);  WRITELN;
       WRITE('Recorrido en Preorden:   ');
       Preorden(a);  WRITELN;
       WRITE('Recorrido en Posorden:   ');
       Posorden(a);  WRITELN;
    END;
    {**********************************************************}
    PROCEDURE Evalua(a:apunt);
    {evalua el arbol}
    VAR  resultado:INTEGER;

       { La funcion Opera aparece en el tema de pilas, en un pro-
        grama que tambien evalua una expresion aritmetica, pag. 13}
       {**********************************************************}
       FUNCTION Opera(simb:CHAR; opnd1,opnd2:INTEGER):INTEGER;
       {de acuerdo al operador simb, se ejecuta la operacion y se
        regresa su  valor}
       {**********************************************************}

       FUNCTION EvaluaArbol(a:apunt):INTEGER;
       VAR
          opnd1, opnd2:INTEGER;
          simb:CHAR;
       BEGIN
          IF a^.inf IN ['0'..'9'] THEN  {es operando}
             EvaluaArbol:=ord(a^.inf)-ord('0')
```

```
            ELSE
               BEGIN
               (es operador, se evaluan subarboles izq. y der.)
                  opnd1:=EvaluaArbol(a^.hijoizq);
                  opnd2:=EvaluaArbol(a^.hijoder);
                  simb:=a^.inf;
                  EvaluaArbol:=Opera(simb,opnd1,opnd2);
               END;
         END;
                      (* inicia procedimiento Evalua *)
      BEGIN
         WRITE('El resultado de la evaluacion es  ');
         resultado:=EvaluaArbol(a);
         WRITELN(resultado);
      END;
(*******************************************************************)
                      ( Programa Principal )
BEGIN
   IniciayLee(ce);
   arbol:=FormaArbol(ce);
   Recorre(arbol);
   Evalua(arbol);
END.
```

```
Escribe la expresion infija a tratar:
(3+2*4)$((7-5)*2)
Recorrido en Inorden:  (3+2*4)$((7-5)*2)
Recorrido en Preorden:  $+3*24*-752
Recorrido en Posorden:  324*+75-2*$
El resultado de la evaluacion es  14641
```

```
Escribe la expresion infija a tratar:
(((6-(3+2))*(3+8/2))$2+3)
Recorrido en Inorden:  ((6-(3+2))*(3+8/2))$2+3
Recorrido en Preorden:  +$*-6+32+3/8223
Recorrido en Posorden:  632+-382/+*2$3+
El resultado de la evaluacion es  52
```

```
Escribe la expresion infija a tratar:
((3+9)-4+((3*5)+9))
Recorrido en Inorden:  3+9-4+3*5+9
Recorrido en Preorden:  +-+394+*359
Recorrido en Posorden:  39+4-35*9++
El resultado de la evaluacion es  32
```

Otra estructura de nodos en Arboles Binarios y su recorrido.

Presentamos la estructura siguiente:
Cada nodo tiene un apuntador a su papá, además de los apuntadores de hijo izquierdo e hijo derecho.

Mostramos la implementación de este árbol binario para recorrerlo en inorden.

```
TYPE
   apunt=^nodotipo;
   nodotipo=RECORD
               inf:inftipo;
               hijoizq:apunt;
               hijoder:apunt;
               papá:apunt;
            END;

FUNCTION ConstruyeArbol(x:inftipo):apunt;
VAR   p:apunt;
BEGIN
   NEW(p);
   p^.hijoizq:=NIL;
   p^.hijoder:=NIL;
   p^.papá:=NIL;
   ConstruyeArbol:=p;
END;

PROCEDURE PonIzq(p:apunt; x:inftipo);
(crea un hijo izquierdo de un nodo)
VAR   q:apunt;
BEGIN
   IF p^.hijoizq <> NIL THEN
      WRITELN('Error, operación ConjuntoIzq ilegal')
   ELSE
      BEGIN
         q:=ConstruyeArbol(x);
         p^.hijoizq:=q;
         q^.papá:=p;
      END;
END;

PROCEDURE PonDer(p:apunt; x:inftipo);
(crea un hijo derecho de un nodo)
VAR   q:apunt;
BEGIN
   IF p^.hijoder <> NIL THEN
      WRITELN('Error, operación ConjuntoDer ilegal')
   ELSE
      BEGIN
         q:=ConstruyeArbol(x);
         p^.hijoder:=q;
         q^.papá:=p;
      END;
END;
```

Recorrido en inorden:

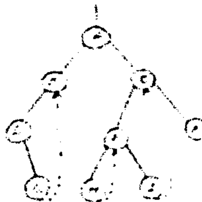
```
PROCEDURE Inorden(árbol:apunt);
VAR
   p,q:apunt;
   fin:BOOLEAN;
BEGIN
   p:=árbol;
   fin:=FALSE;
   REPEAT
      q:=NIL;
      {se recorren los enlaces izquierdos}
      WHILE p <> NIL DO
         BEGIN
            q:=p;
            p:=p^.hijoizq;
         END;
      IF q <> NIL THEN
      BEGIN
         WRITELN(q^.inf);
         p:=q^.hijoder;
         WHILE p=NIL DO
            IF q^.papá <> NIL THEN
            BEGIN
               p:=q^.papá;
               IF q=p^.hijoizq THEN
                  WRITELN(p^.inf)
               ELSE
                  BEGIN
                     WHILE q=p^.hijoder AND p^.papá <> NIL DO
                        BEGIN
                           q:=p;
                           p:=q^.papá;
                        END;
                     IF p^.papá <> NIL THEN
                        WRITELN(p^.inf)
                     ELSE
                        fin:=TRUE;
                  END;
               q:=p;
               p:=q^.hijoder;
            END;
      END
      ELSE
         fin:=TRUE;
   UNTIL fin;
END;
```

ARBOLES BINARIOS HILVANADOS

Se construyen árboles binarios hilvanados con la finalidad de que el recorrido de estos árboles sea de manera más eficiente, esto es, sin usar pilas o recursividad.

En el recorrido de inorden en árboles binarios, la pila es necesaria para ir guardando cada (apuntador al nodo que se va recorriendo hacia abajo en las ramificaciones de la izquierda hasta que se alcanza un apuntador nulo, siendo el elemento superior de la pila el apuntador p antes de que llegue a ser nulo. Si se emplea un apuntador auxiliar q en una etapa anterior a p, el valor de q se puede utilizar directamente y no es necesario sacar de la pila. Otro caso en el cual p resulta nulo es cuando se ha alcanzado un nodo con un subárbol derecho que está vacío; en este momento hubiéramos perdido el camino si no fuera por la pila cuyo elemento superior apunta al nodo cuyo subárbol izquierdo ha sido recorrido. Si se usa en lugar del apuntador nulo un apuntador al nodo que estaría en la parte superior de la pila en ese momento del algoritmo, ya no sería necesario tener la pila, puesto que el nodo directamente apunta a su sucesor en inorden. Este apuntador se define como **hilvanado**, y debe ser diferenciado de un apuntador de árbol, el cual sirve para enlazar un nodo a su subárbol izquierdo o derecho.

Los árboles que se construyen con hilvanes reemplazando a los apuntadores nulos en los nodos que tienen subárboles derechos vacíos, excepto en el nodo que está más a la derecha del árbol (éste posee un apuntador derecho nulo, ya que no presenta ningún sucesor de inorden), son denominados **árboles binarios hilvanados derechos**.



Un Arbol Binario Hilvanado Derecho

Así como hemos definido árbol binario hilvanado derecho de inorden, se puede definir **árbol binario hilvanado izquierdo de inorden** como aquel en el que se altera cada apuntador izquierdo para contener un hilván al nodo de inorden predecesor. Igualmente definimos **árbol** binario hilvanado como un árbol binario que es tanto hilvanado izquierdo como derecho.

OPERACIONES BASICAS

Son las mismas operaciones que en árboles binarios, con ciertos cambios para poder manejar los hilvanes.

La función ConstruyeRaíz(x) crea un árbol binario de un solo nodo con un campo de información x; su hijo derecho es un apuntador hilvanado a NIL, su hijo izquierdo es un apuntador a NIL y retorna un apuntador a este nodo. Esta operación se utiliza

solamente en la creación de la raíz del árbol. La operación ConjuntoIzq(p,x) es análoga a la de árboles binarios, con la diferencia de que el hijo que se crea tiene como hijo derecho apuntando a su papá el nodo p, siendo éste su sucesor en inorden. En la operación ConjuntoDer(p,x), el hijo derecho que se crea del nodo p tiene como hijo derecho apuntando a su sucesor de inorden, éste es el sucesor anterior del nodo p.

IMPLEMENTACION DE ARBOLES HILVANADOS

La implementación se lleva a cabo a través de un arreglo o mediante estructuras dinámicas, así como en la implementación de árboles binarios.

Para implementar un árbol binario hilvanado derecho de inorden con estructura dinámica es necesario agregar un campo booleano adicional a cada nodo del árbol para indicar si su apuntador derecho es de tipo hilvanado.

Por tanto un nodo se define como sigue:

```
TYPE
   apunt=^nodotipo;
   nodotipo=RECORD
               inf:inftipo;
               hijoizq:apunt;
               hijoder:apunt;
               derecho:BOOLEAN {es verdadero si hijo  derecho}
   END;                     {es un apuntador hilvanado no nulo}
```

Operaciones basicas:

```
FUNCTION ConstruyeRaiz(x:inftipo):apunt;
VAR  p:apunt;
BEGIN
   NEW(p);
   p^.inf:=x;
   p^.hijoizq:=NIL;
   p^.hijoder:=NIL;
   p^.derecho:=FALSE;
   ConstruyeRaiz:=p;
END;

PROCEDURE ConjuntoIzq(p:apunt; x:inftipo);
VAR  q:apunt;
BEGIN
   IF p=NIL THEN Error('inserción vacía')
   ELSE
      IF p^.hijoizq <> NIL THEN Error('insercion no válida')
      ELSE
         BEGIN
            NEW(q);
            q^.inf:=x;
            p^.hijoizq:=q;
            q^.hijoder:=p;   {sucesor inorden del nodo q es p}
            q^.derecho:=true; {es apuntador hilvanado}
            q^.hijoizq:=NIL;
         END;
END;
```

```
PROCEDURE Conjuntoder(p:apunt; x:inftipo);
VAR
   q,r:apunt;  {apuntadores auxiliares}
BEGIN
   IF p=NIL THEN Error('inserción vacía')
   ELSE
      IF p^.derecho=false AND p^.hijoder <> NIL THEN
         Error('inserción no válida')
      ELSE
         BEGIN
            NEW(q); q^.inf:=x;
            r:=p^.hijoder;     {se guarda sucesor inorden de p}
            p^.hijoder:=q;
            p^.derecho:=false;     {p^.hijoder no es  hilvanado}
            q^.hijoizq:=NIL;
           {sucesor  inorden de q es el sucesor anterior de  p}
            q^.hijoder:=r;
            IF r <> NIL THEN
               q^.derecho:=true;      {p^.hijoder  es hilvanado}
            ELSE  q^.derecho:=FALSE;
         END;
END;
```

En cuanto a recorridos, escribiremos uno de ellos, el recorrido en inorden.

```
PROCEDURE Inorden(arbol:apunt);
VAR
   p,q:apunt;
BEGIN
   p:=árbol;
   REPEAT
      q:=NIL;
      WHILE  p <> NIL DO  {se recorre  ramificación  izquierda}
         BEGIN
            q:=p;
            p:=p^.hijoizq;
         END;
      IF q <> NIL THEN
         BEGIN
            WRITELN(q^.inf);
            p:=q^.hijoder;
            WHILE q^.derecho DO  {se retrocede}
               BEGIN
                  WRITELN(p^.inf);
                  q:=p;
                  p:=q^.hijoder;
               END;
         END;
   UNTIL q=NIL
END;
```

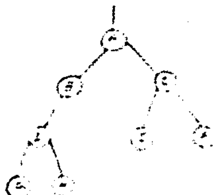
ARBOL BINARIO CASI COMPLETO

Este árbol se implementa de manera sencilla, debido a que no se especifican los enlaces de los nodos en una forma explícita utilizando los campos de padre, hijo izquierdo o hijo derecho.

Arbol binario estrictamente es aquel en el que cada nodo que no es hoja tiene hijos tanto izquierdo como derecho.

Arbol binario casi completo se define como un árbol estrictamente binario para el cual existe un entero k no negativo tal que:

1. Cada hoja en el árbol está a un nivel k o a un nivel k+1.
2. Si un nodo en el árbol tiene un descendiente derecho al nivel k+1 entonces todos sus descendientes izquierdos que son hojas están también al nivel k+1.



Un Arbol Binario Casi Completo

Si el número de hojas en tal árbol es n, entonces el número total de nodos es 2n-1. La importancia de este hecho es que la cantidad de almacenamiento que requiere este árbol es conocida y puede ser declarada.

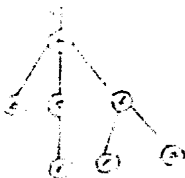
La implementación de estos árboles se lleva a cabo a través de un arreglo de tal manera que el nodo n en el arreglo es el padre implícito de los nodos 2n y 2n+1 del arreglo.

ARBOLES

DEFINICION

Consideramos árboles en forma general.

Un **árbol** es un conjunto finito o vacío de nodos en el que un nodo es denominado la raíz y los nodos restantes son partidos en mayor o igual a cero subconjuntos ajenos, cada uno de los cuales es en sí mismo un árbol.



Un Arbol

Si n1 es la raíz de un árbol y n2 es la raíz del árbol que depende directamente de n1, entonces se dice que n1 es el padre de n2 y n2 es el hijo de n1. Dos nodos que tienen el mismo padre son hermanos.

La definición de un árbol no hace ninguna distinción entre los subárboles de un árbol general, como es el caso de un árbol binario, donde se hace distinción entre los subárboles izquierdo y derecho.

Un **árbol ordenado** es definido como un árbol en el cual los subárboles de cada nodo forman un conjunto ordenado. Es decir, en un árbol ordenado hablamos del primero, segundo o último hijos de un nodo en particular. El primer hijo de un nodo en un árbol ordenado es llamado el hijo mayor de ese nodo y el último es denominado el menor.

En lo subsecuente para referirnos a árboles ordenados utilizaremos solamente la palabra árbol.

OPERACIONES BASICAS

Al construir un arbol, una de las operaciones que se utilizan frecuentemente es ConjuntoHijos(p,lista), la cual recibe un apuntador p de un nodo de un árbol que no tiene hijos y un apuntador (llamado lista) a la lista de nodos encadenados a través de un campo denominado hermano. ConjuntoHijos establece los nodos de la lista como los hijos del nodo apuntado por p en el arbol.

Otra operación común es AñadeHijo(p,x), donde p apunta a un nodo en el arbol y se desea agregar un nodo que contiene información x, como el hijo menor del nodo apuntado por p.

Con respecto a la operación de recorrido de un árbol, es análoga a los recorridos en árboles binarios, donde los apuntadores hijo y hermano toman el papel de hijo izquierdo e hijo derecho, respectivamente.

IMPLEMENTACION DE ARBOLES EN PASCAL

Se implementa ya sea a través de un arreglo o de estructuras dinámicas, de igual forma que las listas.

En los árboles cada nodo puede tener un número variable de hijos, por lo tanto una alternativa es encadenar todos los hijos de un nodo por medio de una lista lineal a través de un apuntador llamado hermano.

Al usar la implementación dinámica, un árbol se puede declarar de la siguiente manera:

```
 TYPE
     apunt=^nodotipo;
     nodotipo=RECORD
                  inf:inftipo;
                  hijo:apunt;
                  hermano:apunt
              END;
```

Las operaciones básicas son:

```
PROCEDURE ConjuntoHijos(p,lista: apunt);
BEGIN
   IF p=NIL  THEN
      WRITELN('Error, inserción no válida')
   ELSE
      IF p^.hijo <> NIL THEN
         WRITELN('Error inserción no válida')
      ELSE  p^.hijo:=lista;
END;

PROCEDURE AñadeHijo(p:apunt; x:inftipo);
(agrega un nodo como el hijo menor del nodo apuntado por p)
VAR
   q,r:apunt;
BEGIN
   IF p=NIL THEN
      WRITELN('Error, inserción no válida')
   ELSE
      BEGIN
         r:=NIL; (r es un apuntador a un nodo más atrás que q)
         q:=p^.hijo; (se recorre la lista de hijos de p por q)
         WHILE q <> NIL DO (hasta llegar al hijo menor)
            BEGIN
               r:=q;  (r puede ser NIL si p no tiene hijos)
               q:=q^.hermano;
            END;
         NEW(q);(se forma el nodo hijo, el cual se asigna a p)
         q^.inf:=x;
         q^.hermano:=NIL;
         IF r=NIL  THEN  (p no tiene hijos)
            p^.hijo:=q
         ELSE
            r^.hermano:=q;
      END;
END;
```

Una aplicación de los árboles.

Los árboles tienen varias aplicaciones. Una de ellas es en los juegos.

Presentamos el juego denominado "el imposible". Este consiste en recibir una tabla inicial y final de n*n con n*n-1 elementos. Se trata de encontrar el camino (los movimientos posibles) para llegar del estado inicial al final.

Mostraremos con un ejemplo:

```
(283)                                123
(164) ---> ---> ---> --->            804
(705)                                765

 Inicial                             Final
```

De acuerdo a las características del problema a resolver, los nodos del árbol pueden cambiar en su estructura con respecto a la forma que hemos definido anteriormente.

Los nodos que utilizaremos en este árbol tienen dos campos, que son la información (una tabla o matriz) y un apuntador a su papá; el nodo inicial tiene apuntador a NIL.

ALGORITMO

1. Colocamos el nodo inicial en una cola de apuntadores.
2. Mientras no esté vacía la cola y no hayamos llegado al nodo final, continuamos.
3. Retiramos un nodo de la cola; sea n.
4. Expandemos el nodo n. El orden en que se expanden los nodos corresponde a la llamada búsqueda horizontal ("breadth first"). Esto es generar todos sus sucesores, si no hay sucesores ir a (2).
5. Ver si alguno de estos sucesores es el nodo final.
   - Si es el final terminamos, obteniendo la solución al recorrer los nodos a través de sus apuntadores a sus nodos padres.
   - Si no es el final poner los sucesores en la cola y poner apuntadores de estos hacia sus nodos padres e ir a (2).

Este juego inicia con un estado diferente al final.

```
PROGRAM Juego;
(Programacion del juego denominado "el imposible")
(utilizando una estructura de arbol)
CONST
   n=3;          (la tabla es de 3*3)
   maxcola=100; (maximo de elementos de la cola circular)
TYPE
   apunt=^nodotipo;
   inftipo=ARRAY[1..n,1..n] OF INTEGER;
   nodotipo=RECORD
               inf:inftipo;
               papa:apunt
            END;
   cola=RECORD
           elemento:ARRAY[1..maxcola] OF apunt;
           frente,fondo:0..maxcola;
           llenando:BOOLEAN
        END;
VAR
   inicio,fin:apunt;    (apuntadores  inicial y final del juego)
   solucion:BOOLEAN;    (verdadero si se llego a la tabla final)
   abierto:cola;  (cola de apuntadores a los estados del juego)
   i,j:INTEGER;    (variables para recorrer las tablas o matriz)
   p:apunt;         (apuntador al nodo que  se retira de la cola)

   (****************************************************************)
   PROCEDURE IniciayLee;
   (inicializa la  cola llamada abierto e inicializa el juego
    leyendo su estado inicial y final)
   VAR mataux:inftipo;
       apuntaux:apunt;
       k:INTEGER;
   BEGIN
      WITH abierto DO
         BEGIN
            frente:=0;  fondo:=0;
            llenando:=FALSE;
         END;
      solucion:=FALSE; (inicia con estado diferente al final)
      FOR k:=1 TO 2 DO
         BEGIN
            NEW(apuntaux);
            IF k=1 THEN
               BEGIN
                  WRITELN('Escribe una matriz de ',n,'*',n);
                  WRITELN('que representa el inicio del juego');
                  inicio:=apuntaux;
               END
            ELSE
               BEGIN
                  WRITELN;
                  WRITELN('Escribe solucion final del juego');
                  fin:=apuntaux;
               END;
            FOR i:=1 TO n DO
               BEGIN
```

```
              FOR j:=1 TO n DO
                 READ(mataux[i,j]);
              WRITELN;
           END;
        apuntaux^.inf:=mataux;
        apuntaux^.papa:=NIL;
     END;
   WRITELN; WRITELN;
END;

(Procedimientos para la manipulacion de una cola circular)
(*************************************************************)
FUNCTION Lleno(c:cola):BOOLEAN;
BEGIN
   Lleno:=((c.llenando)AND(c.frente=(c.fondo+1)MOD maxcola));
END;
(*************************************************************)
FUNCTION Vacio(c:cola):BOOLEAN;
BEGIN
   Vacio:=NOT(c.llenando)AND(c.frente=(c.fondo+1)MOD maxcola);
END;
(*************************************************************)
FUNCTION Retira(VAR c:cola):apunt;
BEGIN
   IF NOT Vacio(c) THEN
      BEGIN
         Retira:=c.elemento[c.frente];
         IF c.frente=maxcola THEN c.frente:=1
         ELSE
            c.frente:=c.frente+1;
         c.llenando:=FALSE;
      END
   ELSE  WRITELN('cola vacia');
END;
(*************************************************************)
PROCEDURE Inserta(VAR c:cola; X:apunt);
BEGIN
   IF NOT Lleno(c) THEN
      BEGIN
         IF c.fondo=maxcola THEN c.fondo:=1
         ELSE
            c.fondo:=c.fondo+1;
         c.elemento[c.fondo]:=X;
         c.llenando:=TRUE;
         IF c.frente=0 THEN
            c.frente:=1;
      END
   ELSE  WRITELN('cola llena');
END;
(*************************************************************)
FUNCTION Compara(p,q:apunt):BOOLEAN;
(Compara  dos matrices, si son iguales la funcion  regresa el
 valor de verdadero, en caso contrario el de falso)
VAR
   a,b:inftipo; (variables auxiliares)
BEGIN
```

```
    a:=p^.inf;  b:=q^.inf;
    Compara:=TRUE;
    FOR i:=1 TO n DO
       FOR j:=1 TO n DO
          IF a[i,j] <> b[i,j] THEN
             Compara:=FALSE;
END;
{***********************************************************}
PROCEDURE ExpandeNodo(VAR p:apunt);
{Expande  un  nodo.  Esto es generando  sus hijos,  antes de
 insertarlos en la cola  compara que no sean el nodo final y
 que no esten en la trayectoria}
VAR
    k,ii,jj:INTEGER;
    q,R:apunt;
    mov:BOOLEAN;      {verdadero si genera hijo(nodo)}
    esta:BOOLEAN;     {verdadero si hijo esta en trayectoria}
    a,b:inftipo;
    {***********************************************************}
    PROCEDURE GeneraMov(k:INTEGER;VAR q:apunt;VAR mov:BOOLEAN);
    {Genera los movimientos o hijos posibles}
    BEGIN
       b:=a;
       mov:=TRUE;
       CASE k OF
          1:    IF jj <> 1 THEN
                   BEGIN
                      b[ii,jj]:=a[ii,jj-1];
                      b[ii,jj-1]:=0;
                   END
                ELSE
                   mov:=FALSE;
          2:    IF ii <> 1 THEN
                   BEGIN
                      b[ii,jj]:=a[ii-1,jj];
                      b[ii-1,jj]:=0;
                   END
                ELSE
                   mov:=FALSE;
          3:    IF jj <> n THEN
                   BEGIN
                      b[ii,jj]:=a[ii,jj+1];
                      b[ii,jj+1]:=0;
                   END
                ELSE
                   mov:=FALSE;
          4:    IF ii <> n THEN
                   BEGIN
                      b[ii,jj]:=a[ii+1,jj];
                      b[ii+1,jj]:=0;
                   END
                ELSE
                   mov:=FALSE;
       END;
       IF mov THEN
          BEGIN
```

```
            NEW(q);
            q^.inf:=b;
         END;
   END;
                  (* inicia procedimiento ExpandeNodo *)
BEGIN
   esta:=FALSE; a:=p^.inf;
   (encuentra el elemento vacio (el cero) en la tabla)
   FOR i:=1 TO n DO
      FOR j:=1 TO n DO
         IF a[i,j]=0 THEN
            BEGIN
               ii:=i;  jj:=j;
            END;
   k:=1;
   WHILE ((k <= 4) AND NOT (solucion)) DO
      BEGIN
         GeneraMov(k,q,mov);
         IF mov THEN
            IF NOT Compara(q,fin) THEN
               BEGIN
                  R:=p;
                  esta:=FALSE;
                  WHILE ((R <> NIL)  AND NOT esta) DO
                     BEGIN
                        IF Compara(R,q) THEN
                           esta:=TRUE;
                        R:=R^.papa;
                     END;
                  IF NOT esta THEN
                     BEGIN
                        Inserta(abierto,q);
                        q^.papa:=p;
                     END;
               END
            ELSE
               BEGIN
                 solucion:=TRUE;
                 q^.papa:=p;
                 p:=q;
              END;
         k:=k+1;
      END;
END;
(***************************************************************)
PROCEDURE EscribeSolucion;
(Utilizamos un arreglo para ir guardando los apuntadores a los
 nodos que vamos recorriendo  hacia atras y escribimos la solu-
 cion en orden contrario a los indices del arreglo)
VAR
   Resultado:ARRAY[1..100] OF apunt;
   k,l:INTEGER;
   a:inftipo;
BEGIN
   WRITELN('Movimientos para obtener la solucion');
   WRITELN;
```

```
        k:=1;
        WHILE p <> NIL DO
           BEGIN
              Resultado[k]:=p;
              p:=p^.papa;
              k:=k+1;
           END;
         FOR i:=1 TO n DO
            BEGIN
               FOR l:=k-1 DOWNTO 1 DO
                  BEGIN
                     a:=Resultado[L]^.inf;
                     FOR j:=1 TO n DO
                        WRITE((a[i,j]));
                     IF ((i=2) AND (l<>1)) THEN  WRITE(' ----> ')
                     ELSE  WRITE('        ');
                  END;
               WRITELN;
         END;
    END;
 {*************************************************************}
                     (Programa Principal)

 BEGIN
    IniciayLee;
    Inserta(abierto,inicio);
    WHILE(NOT Vacio(abierto) AND NOT(solucion)) DO
       BEGIN
          p:=Retira(abierto);
          ExpandeNodo(p);
       END;
    IF solucion THEN
       EscribeSolucion;
 END.
```

```
Escribe una matriz de 3*3
que representa el inicio del juego
283
164
705

Escribe solucion final del juego
123
804
765


Movimientos para obtener la solucion

283        283        203        023        123        123
164 ---->  104 ---->  184 ---->  184 ---->  084 ---->  804
705        765        765        765        765        765
```

GRAFICAS

Una **gráfica** es un conjunto no vacío de nodos (vertices) y un conjunto de arcos. Cada arco se especifica por una pareja de nodos.

Definición de algunos términos asociados con gráficas.

Una **gráfica dirigida** es aquella en la que los arcos se especifican por una pareja ordenada.

Un nodo n es **adyacente** al nodo m si existe un arco de m a n.

Una **gráfica con factor de peso** es aquella en la cual un número está asociado con un arco. El numero asociado con un arco es denominado su **factor de peso**.

Un **camino de longitud k** desde el nodo a hasta el nodo b se define como una secuencia de k+1 nodos n1,n2,...nk+1 tal que n1=a, nk+1=b y ni+1 es adyacente a ni para todo i entre 1 y k.

Un camino de un nodo a sí mismo es denominado un **ciclo**. Si una gráfica contiene un ciclo, se denomina **cíclica**; en caso contrario se llama **acíclica**.

OPERACIONES BASICAS

Identificaremos algunas operaciones basicas que son útiles al manejar gráficas. La operación Union(a,b) agrega un arco del nodo a al nodo b si éste no existe. UnionPeso(a,b,x) agrega un arco de a a b con el factor de peso x en una gráfica con factor de peso. Quita(a,b) y QuitaPeso(a,b,x) remueven un arco desde a hasta b si este existe (QuitaPeso tambien coloca en x su factor de peso). Omitiremos las operaciones de agregar o retirar nodos de una gráfica. La función adyacente(a,b) retorna el valor verdadero si b es adyacente a a y falso en caso contrario.

IMPLEMENTACION DE GRAFICAS

La forma de representar gráficas es a través de un arreglo (matriz) o de estructuras dinámicas.

Implementación con matriz.

Asumimos que el número de nodos en la gráfica es constante (es decir, que se pueden agregar o quitar arcos pero no cambiar el número de nodos) y que existe un orden entre los nodos de la gráfica en el sentido de que un nodo particular es llamado el primer nodo, otro el segundo y así sucesivamente. La matriz de representación de la gráfica depende de la ordenación de los nodos. Frecuentemente los nodos de una gráfica son numerados de 1 a maxnodos y no se les da ninguna información a ellos. Ademas podemos estar interesados en la existencia de arcos pero no en algún factor de peso u otra información relacionada con ellos. En estos casos la gráfica se puede declarar en forma simple, como:

```
CONST
   maxnodos=50;
TYPE
   apunt=1..maxnodos;
   matrizady=ARRAY[apunt, apunt] OF BOOLEAN;
VAR
   ady:matrizady;
```

El arreglo bidimensional ady[apunt,apunt] es denominado matriz adyacente. El elemento i,j es verdadero si existe un arco de i a j.

Las operaciones basicas son:

```
PROCEDURE Union(VAR ady:matrizady; nodo1,nodo2:apunt);
{agrega un arco desde nodo1 hasta nodo2}
BEGIN
   ady[nodo1,nodo2]:=TRUE;
END;

PROCEDURE Quita(VAR ady:matrizady; nodo1,nodo2:apunt);
{elimina arco desde nodo1 hasta nodo2 si este existe}
BEGIN
   ady[nodo1,nodo2]:=FALSE;
END;

FUNCTION Adyacente(ady:matrizady; nodo1,nodo2:apunt):BOOLEAN;
{prueba si existe un arco desde nodo1 hasta nodo2, esto es que
nodo2 es adyacente con nodo1}
BEGIN
   IF ady[nodo1,nodo2] THEN
      Adyacente:=ady[nodo1,nodo2]
   ELSE
      Adyacente:=FALSE;
END;
```

UNA APLICACION DE LAS GRAFICAS

Resolveremos el siguiente ejercicio mediante las dos formas de implementación.

Dado un conjunto de n vértices y un conjunto de arcos, encontrar todos los caminos (matriz de caminos) de longitudes 1..n-1 entre los nodos.

La matriz adyacente (ady) representa todos los caminos de longitud 1. Para encontrar los caminos de longitud 2 se multiplica ady consigo misma -la multiplicación numérica es remplazada por la conjunción (la operación AND) y la suma es substituida por la disyunción (la operación OR)-. Por consiguiente se dice que ady2 es el producto booleano de ady consigo misma.

De manera análoga se define ady3 (camino de longitud 3 de la matriz) como el producto booleano de ady2 con ady1. Donde ady3[i,j] es verdadero si y solamente si existe un camino de longitud 3 desde i hasta j.

En general, para calcular un camino de longitud L de la matriz se forma el producto booleano de los caminos de la matriz de longitud L-1 con la matriz adyacente.

Deseamos conocer si un camino de longitud k o menor existe entre los nodos i y j. Si este camino existe debe ser de longitud 1,2...c k, y el valor de la siguiente expresión tiene que ser verdadero:

ady[i,j] OR ady2[i,j] OR ... OR adyk[i,j]

La matriz formada al aplicar la operación OR a las matrices ady, ady2... y adyk es denominada la matriz de caminos.

Escribimos en PASCAL una rutina que calcula la matriz de caminos, dada la matriz adyacente. Esta rutina utiliza otra rutina auxiliar Prod (a,b,c), la cual hace el arreglo c igual al producto booleano de a y b.

```
PROCEDURE MatrizCamino(ady:matrizady; VAR camino:matrizady);
VAR
   i,j,k:apunt;
   nuevoprod,adyprod:matrizady;

   PROCEDURE Prod(a,b:matrizady; VAR c:matrizady);
   VAR
      val:BOOLEAN;
      i,j,k:INTEGER;
   BEGIN
      FOR i:=1 TO maxnodos DO      (* recorrido de las filas *)
         FOR j:=1 to maxnodos DO  (* recorrido de las columnas *)
            BEGIN                 (* calcula c[i,j] *)
               val:=FALSE;
               FOR k:=1 TO maxnodos do
                  val:=val OR (a[i,k] AND b[k,j]);
               c[i,j]:=val;
            END;
   END;
```

```
BEGIN   (de MatrizCamino)
   adyprod:=ady;
   camino:=ady;
   FOR i:=1 TO maxnodos-1 DO
  (* i representa el número de veces que ady ha sido multiplicada
    por sí misma para obtener adyprod. En este punto, camino
    representa todos los caminos de longitud i o menores *)
   BEGIN
      Prod(adyprod,ady,nuevoprod);
      FOR j:=1 TO maxnodos DO
         FOR k:=1 TO maxnodos DO
            camino[j,k]:=camino[j,k] OR nuevoprod[j,k];
         adyprod:=nuevoprod;
   END;
END;
```

El método que se ha descrito es un poco ineficiente. Presentamos en seguida una implementación más efectiva para calcular la matriz (camino).

Definamos la matriz caminok tal que caminok[i,j] es igual a verdadero si y solamente si existe un camino desde el nodo i hasta el nodo j que no pasa a través de algunos de los nodos numerados más altos que k (excepto, posiblemente, para i y j).

Obtenemos el valor de caminok+1 a partir del valor de caminok. Para cualquier i y j tal que caminok[i,j] es igual a verdadero, entonces caminok+1 debe ser igual a verdadero. La única situación en la cual caminok+1[i,j] puede ser igual a verdadero cuando caminok[i,j] es igual a falso sucede si hay un camino desde i hasta j pasando a través del nodo k+1, pero no existe camino desde i hasta j pasando únicamente a través de los nodos desde 1 hasta k. Esto significa que debe haber un camino de i a k+1 pasando a través de los nodos 1 hasta k y en forma similar un camino desde k+1 hasta j. Es decir, caminok+1[i,j]=verdadero si y solamente si se cumple una de las siguientes condiciones:

1.- caminok[i,j]=verdadero o
2.- caminok[i,k+1]=verdadero y caminok[k+1,j]=verdadero.

Este método de encontrar la matriz de caminos es conocido como Algoritmo de Warshall, en honor a su descubridor. La rutina siguiente calcula la matriz de caminos mediante el empleo de este algoritmo.

```
PROCEDURE Warshall(ady:matrizady; VAR camino:matrizady);
VAR
   i,j,k:INTEGER;
BEGIN
   camino:=ady;
   FOR k:=1 TO maxnodos DO
      FOR i:=1 TO maxnodos DO
         FOR j:=1 TO maxnodos DO
            camino[i,j]:=(camino[i,j] OR
            (camino[i,k] AND camino[k,j]));
END;
```

Debido a que la matriz de adyacencia generalmente se presenta como una matriz dispersa, damos la implementación de la matriz con listas multiencadenadas ya que es la más adecuada.

Para permitir el fácil acceso a cualquier fila o columna de la matriz, dejamos que aparezca cada elemento diferente de cero enlazado por dos ligas, una para su fila y otra para su columna. Por lo tanto cada nodo contiene dos apuntadores, uno al elemento siguiente en su fila y otro al elemento siguiente en su columna. Además cada nodo contiene campos con el número de su fila, el número de su columna y el valor de su elemento. Este tipo de nodo se puede definir como:

```
CONST
   m=...;
   n=...;
TYPE
   numren=1...m;
   numcol=1...n;
   apunt=^nodotipo;
   nodotipo=RECORD
               ren:numren;    {numero de renglon}
               col:numcol;    {numero de columna}
               val:INTEGER;   {valor del nodo}
               rensig:apunt;  {elemento sig. en la misma columna}
               colsig:apunt   {elemento sig. en el mismo renglón}
END;
```

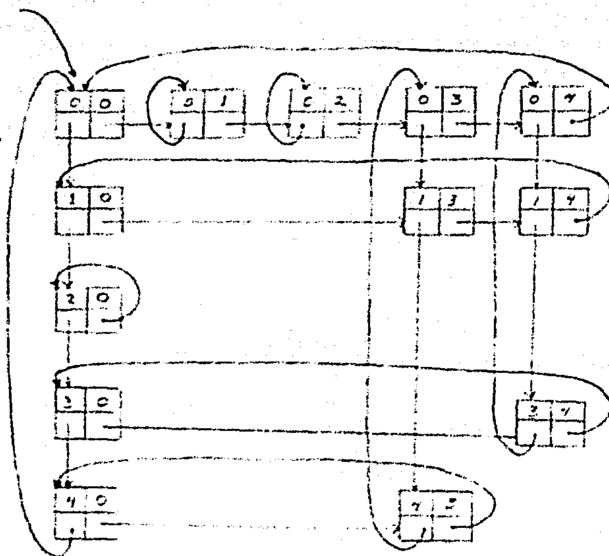
Al igual que con cualquier lista encadenada, se debe seguir algún método para accesar el primer elemento de cada lista que representa la matriz.

Consideramos el método que consiste en mantener una columna arbitraria (columna 0) que contenga tantos elementos como filas existan. Cada elemento en esta columna arbitraria apunta al primer elemento de su fila respectiva. Igualmente, hay una fila arbitraria (fila 0) que contiene tantos elementos como columnas tenga la matriz. Cada elemento en esta fila apunta al primer elemento de su respectiva columna. Cada nodo de estas listas arbitrarias sirve como un nodo de encabezamiento para la lista de fila o de columna. Estos nodos de encabezamiento se pueden reconocer por un cero en el campo de columna o de fila. Por tanto ahora los tipos numren y numcol pueden tomar valores de 0...m y 0...n, respectivamente. Puesto que cada fila o columna tiene un encabezamiento, se pueden conservar como una lista circular. Un apuntador externo puede apuntar a un elemento arbitrario en la fila 0, columna 0; este elemento sirve como nodo de encabezamiento de la matriz. Dado este apuntador es posible encontrar cualquier nodo de la matriz.

En la siguiente página mostramos con un dibujo la representación de la matriz.

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

Matriz en forma de arreglo



Matriz utilizando asignación multiencadenada

```
PROGRAM Warshall;
{ Se calcula la matriz camino de una grafica, y se implementa con
  listas multiencadenadas}
TYPE
   apunt=^nodotipo;
   nodotipo=RECORD       {estructura de cada nodo}
              ren: INTEGER;    {numero de renglon}
              col: INTEGER;    {numero de columna}
              sigren: apunt;   {elemento sgt. en el mismo renglon}
              sigcol: apunt    {elemento sig. en la misma columna}
            END;
VAR
   maxnodos:INTEGER; {numero maximo de nodos de la grafica}
   existe:BOOLEAN;   {auxiliar en la busqueda de un nodo}
   camino:apunt:     {guarda la matriz camino resultante}
   ady:apunt;        {matriz de adyacencia de la grafica}

   {Se omiten las siguientes rutinas, debido a que ya apecen
    al tratar listas multiligadas, paginas 59-61}
   {*************************************************************}
   PROCEDURE InsertaDespues(p:apunt; r,c:INTEGER );
   {inserta un nodo con informacion r y c despues del nodo apun-
    tado por p}
   {*************************************************************}
   FUNCTION CreaCabezas(maxnodos:INTEGER):apunt;
   {esta funcion  crea el renglon y columna de  encabezamiento,
    regresando un apuntador al nodo que los une, siendo este el
    nodo cabeza de la matriz}
   {*************************************************************}
   FUNCTION BuscaRen(r,c:INTEGER; matriz:apunt):apunt;
   {localiza el renglon r, recorre este si hay nodos hasta llegar
   a la columna  mas grande y menor que c, se posiciona en este
   elemento regresando un apuntador a el}
   {*************************************************************}
   FUNCTION BuscaCol(r,c:INTEGER; matriz:apunt):apunt;
   {localiza la columna c, recorre esta si hay nodos hasta llegar
    al renglon  mas grande y menor  que r, se posiciona en este
    elemento regresando un apuntador a el}
   {*************************************************************}
   PROCEDURE InsertaNodo(r,c:INTEGER; matriz:apunt);
   {inserta un nodo en el renglon r y columna c de la matriz}
   {*************************************************************}

   PROCEDURE LeeMatriz(VAR matriz:apunt);
   VAR
      r,c,v:INTEGER:  {renglon, columna y valor respectivamente}
   BEGIN
      WRITE('Escribe el numero total de nodos: ');
      READLN(maxnodos);
      WRITELN('Escribe la matriz de adyacencia por renglones');
      WRITELN:
      matriz:=CreaCabezas(maxnodos);
      FOR r:=1 TO maxnodos DO
         BEGIN
            WRITE(' ':15);
            FOR c:=1 TO maxnodos DO
```

```
          BEGIN
             READ(v);
             IF v <> 0 THEN
                InsertaNodo(r,c,matriz);
          END;
       READLN;
    END; WRITELN;
END;
{*************************************************************}
PROCEDURE BuscaNodo(r,c:INTEGER; matriz:apunt);
{si se encuentra el nodo en el renglon r y columna c entonces
 la variable global llamada existe toma el valor verdadero}
VAR
   aptren,aptcol:apunt;
BEGIN
   aptren:=BuscaRen(r,c,matriz);
   aptcol:=BuscaCol(r,c,matriz);
   IF (aptren=NIL) AND (aptcol=NIL) THEN
      existe:=TRUE
   ELSE
      existe:=FALSE;
END;
{*************************************************************}
PROCEDURE Warshall(ady:apunt; VAR camino:apunt);
VAR
   i,j,k:INTEGER;
BEGIN
   camino:=ady;
   FOR k:=1 TO maxnodos DO
      FOR i:=1 TO maxnodos DO
         FOR j:=1 TO maxnodos DO
            BEGIN
               BuscaNodo(i,j,camino);
               IF NOT existe THEN
                  BEGIN
                     BuscaNodo(i,k,camino);
                     IF existe THEN
                        BuscaNodo(k,j,camino);
                        IF existe THEN
                           InsertaNodo(i,j,camino);
                  END;
            END;
END;
{*************************************************************}
PROCEDURE Escribe(matriz:apunt);
VAR
   r,c:INTEGER;
   p,q:apunt;
BEGIN
   WRITELN('Matriz camino');
   WRITELN('Indica todos los caminos de diferentes longitudes');
   WRITELN('que existen entre los nodos'); WRITELN;
   p:=matriz;
   FOR r:=1 TO maxnodos DO
      BEGIN
         p:=p^.sigren;  q:=p^.sigcol;
```

```
            WRITE(' ':15);
            FOR c:=1 TO maxnodos DO
               IF q^.col=c THEN
                  BEGIN
                     WRITE('1  ');
                     q:=q^.sigcol;
                  END
               ELSE
                  WRITE('0  ');
            WRITELN;
         END;
   END;
   (****************************************************************)
                       { Programa Principal }
BEGIN
   LeeMatriz(ady);
   Warshall(ady,camino);
   Escribe(camino);
END.
```

```
Escribe el numero total de nodos: 5
Escribe la matriz de adyacencia por renglones

                0  0  1  1  0
                0  0  1  0  0
                0  0  0  1  0
                0  0  0  0  1
                0  0  0  1  0

Matriz camino
Indica todos los caminos de diferentes longitudes
que existen entre los nodos

                0  0  1  1  1
                0  0  1  1  1
                0  0  0  1  1
                0  0  0  1  1
                0  0  0  1  1
```

```
PROGRAM Reconoce;
( Otro ejemplo de aplicacion de graficas.
  Reconoce si una cadena de caracteres es aceptada por un automa-
  ta dado, representado por una grafica.)
TYPE
   apunt=^nodotipo;
   inftipo=CHAR;
   nodotipo=RECORD  (estructura de los nodos de la grafica)
                inf:inftipo;
                apunta:apunt;
                sig:apunt;
                fin:BOOLEAN
            END;
VAR
   grafica:apunt;    (apuntador a la grafica)

   (************************************************************)
   FUNCTION CreaCabeza:apunt;
   (crea los nodos de la grafica, regresando un apuntador a dicha
    grafica)
   VAR
      c:CHAR;
      i,n:INTEGER;
      p,q:apunt;
   BEGIN
      WRITE('Cuantos nodos tiene la grafica?: ');
      READLN(n);
      WRITELN; WRITE('Escribe cuales son: ');
      READ(c);
      NEW(q);
      q^.inf:=c;
      q^.apunta:=NIL;
      q^.sig:=NIL;
      q^.fin:=FALSE;
      CreaCabeza:=q;
      FOR i:=2 TO n DO
          BEGIN
             READ(c);
             NEW(p);
             p^.inf:=c;
             p^.apunta:=NIL;
             p^.sig:=NIL;
             p^.fin:=FALSE;
             q^.sig:=p;
             q:=q^.sig;
          END; WRITELN;
   END;
   (************************************************************)
   FUNCTION BuscaNodo(Grafica:apunt; x:CHAR):apunt;
   (busca un nodo en particular, regresando un apuntador)
   VAR
      p:apunt;
      encuentra:BOOLEAN;
   BEGIN
      p:=Grafica;
      encuentra:=FALSE;
```

```
   WHILE ((p <> NIL) AND (NOT encuentra)) DO
        IF p^.inf=x THEN
           BEGIN
              BuscaNodo:=p;
              encuentra:=TRUE;
           END
        ELSE
           p:=p^.sig;
   IF NOT encuentra THEN
      BuscaNodo:=NIL
END;
{*************************************************************}
PROCEDURE CreaArco(p,q:apunt; peso:CHAR);
{este procedimiento acepta dos apuntadores (p y q) a dos nodos
 de encabezamiento  y crea un arco (representado  por un nodo)
 entre ellos, con un factor de peso}
VAR
   r,s:apunt;
   encuentra:BOOLEAN;
BEGIN
   {busca un arco de p a q}
   encuentra:=FALSE;
   s:=NIL;
   r:=p^.apunta;
   WHILE ((r <> NIL) AND (NOT encuentra)) DO
      IF r^.apunta=q THEN
         encuentra:=TRUE
      ELSE
         BEGIN
            s:=r;
            r:=r^.sig;
         END;
      IF NOT encuentra THEN
         {no existe un arco desde p hasta q; se crea}
         BEGIN
            NEW(r);
            r^.apunta:=q;
            r^.sig:=NIL;
            r^.inf:=peso;
            IF s=NIL THEN
               p^.apunta:=r
            ELSE
               s^.sig:=r;
         END;
END;
{*************************************************************}
FUNCTION ConstruyeGrafica:apunt;
{se construye la grafica utilizando las rutinas anteriores}
VAR
   n1,n2,peso:CHAR;
   i,n:INTEGER;
   p,q,grafica:apunt;
BEGIN
   grafica:=CreaCabeza;
   WRITELN; WRITE('Cuantos estados finales hay?: ');
   READLN(n);
```

```
   WRITELN; WRITE('Cuales son?: ');
   FOR i:=1 to n DO
      BEGIN
         READ(n1);
         p:=BuscaNodo(Grafica,n1);
         p^.fin:=TRUE;
      END;
   WRITELN; WRITELN;
   WRITELN('Escribe ternas de informacion por renglon (pareja');
   WRITELN('de nodos y peso); para terminar escribe puntos');
   READLN(n1,n2,peso);
   WHILE n1 <> '.' DO
      BEGIN
         p:=BuscaNodo(Grafica,n1);
         q:=BuscaNodo(Grafica,n2);
         CreaArco(p,q,peso);
         READLN(n1,n2,peso);
      END;
   ConstruyeGrafica:=grafica;
END;
{**************************************************************}
PROCEDURE LeeyReconoce(grafica:apunt);
VAR
   i,n:INTEGER;
   p:apunt;
   c:CHAR;
BEGIN
   WRITELN; WRITE('Cuantas corridas deseas?: ');
   READLN(n);
   FOR i:=1 TO n DO
      BEGIN
         p:=Grafica;
         WRITELN; WRITELN('Escribe la cadena a reconocer;');
         WRITELN('para terminar escribe un punto');
         READ(c);
         WHILE (c='0') OR (c='1') DO
            BEGIN
               p:=p^.apunta;
               WHILE ((p^.inf <> c) AND (p <> NIL)) DO
                  p:=p^.sig;
               IF p=NIL THEN  c:='.'
               ELSE
                  BEGIN
                     p:=p^.apunta;
                     READ(c);
                   END;
            END; WRITELN;
         IF p=NIL  THEN
            WRITELN('   !!! Cadena no reconocida !!!')
         ELSE
            IF p^.fin=TRUE THEN
               WRITELN('   !!! Cadena reconocida !!!')
            ELSE
               WRITELN('!!! Cadena no recoñocica !!!');
   END;
END;
```

```
{*************************************************************}
                    {Programa Principal}
BEGIN
   grafica:=ConstruyeGrafica;
   LeeyReconoce(grafica);
END.
```

```
Cuantos nodos tiene la grafica?: 4

Escribe cuales son: 1 2 3 4

Cuantos estados finales hay?: 1

Cuales son?: 1

Escribe ternas de informacion por renglon (pareja
de nodos y peso); para terminar escribe puntos
1 2 1
1 4 0
2 1 1
2 3 0
3 2 0
3 4 1
4 1 0
4 3 1
. . .

Cuantas corridas deseas?: 2

Escribe la cadena a reconocer;
para terminar escribe un punto
111000110001.
   !!! Cadena reconocida !!!

Escribe la cadena a reconccer;
para terminar escribe un punto
11111000.
   !!! Cadena no reconocida !!!
```

ORDENAMIENTO

Ordenamiento y búsqueda son dos procesos muy comunes en programación. En este tema presentamos algunos algoritmos de ordenamiento y su implementación.

Un **algoritmo de ordenamiento** clasifica los elementos de un conjunto o archivo -en orden ascendente o descendente-, en el que el orden se define de acuerdo con el tipo de datos.

El mejor método de clasificación varía según el conjunto. Depende de factores como el tipo de conjunto, el tamaño, la distribución inicial de sus elementos, la magnitud de éstos y los medios que se disponen para manipular los mismos (aquí consideramos ordenamiento interno, esto es, en memoria principal). Por ello existen diferentes métodos de ordenamiento, que dan una solución adecuada a cada conjunto.

Presentamos la clasificación de los siguientes algoritmos de ordenamiento y su implementación:

1.- Ordenamiento de intercambio
  -Ordenamiento rápido (Quicksort)

2.- Ordenamiento de selección y de árbol
  -Ordenamiento de grupo (Heapsort)

3.- Ordenamiento de inserción
  -Ordenamiento de Shell

4.- Ordenamiento de concatenación
  -Ordenamiento de concatenación (Merge).

Existen muchos otros algoritmos. Por ejemplo, árbol binario (ya presentado en el tema de árboles), burbuja, competencia, inserción simple, cálculo de direcciones y ordenamiento de base (radix).

Todos los algoritmos que aparecen a continuación ordenan una lista de diez elementos (puede aumentarse este número, según sea necesario) organizados en un arreglo. Se ordena la misma lista (en forma ascendente) a fin de poder apreciar el funcionamiento de cada algoritmo.

ORDENAMIENTO RAPIDO (Quicksort)

El ordenamiento a tratar es el de intercambio con partición (ordenamiento rápido). El método Quicksort desarrollado por C.A.R. Hoare tiene el mejor promedio de comportamiento entre todos los métodos de ordenar arreglos.

Considere x como un arreglo y n el número de elementos en el arreglo que han de ser ordenados.

ALGORITMO

El algoritmo consiste en escoger un elemento k (cuya posición final se quiere encontrar) de alguna posición específica dentro del arreglo (k puede ser escogido como el primer elemento tal que k=x[1]) y colocar este elemento k en la posición j, la cual es la correcta dentro del arreglo ordenado, de tal manera que se cumplan las condiciones siguientes:

- Los elementos en posiciones 1 hasta j-1 son menores o iguales a k.
- Los elementos en posiciones j+1 hasta n son mayores o iguales a k.

Si se repite el proceso con los subarreglos x[1] hasta x[j-1] y x[j+1] hasta x[n] y con cualquiera de los subarreglos creados mediante el proceso de iteraciones sucesivas, el resultado final será un archivo ordenado.

ANALISIS

El ordenamiento Quicksort es mejor para archivos en donde las llaves se encuentran aleatoriamente distribuidas y es peor para archivos que están ordenados o casi ordenados.

Si se considera que siempre se elige la mediana del arreglo como la posición apropiada para el elemento k, entonces cada proceso de partición divide al arreglo en dos partes iguales (n, 2(n/2), 4(n/4),..., n(n/n)). De aquí que el número necesario de procesos de partición (pasos) para ordenar el arreglo es $\log_2 n$, y si cada paso requiere n comparaciones podemos decir que el número total de comparaciones para todo el archivo es $n \log n$.

Existe poca probabilidad de que se elija siempre la mediana, sin embargo el rendimiento promedio se mantiene en ese orden cuando la posición del elemento k es aleatoria.

El peor caso es cuando el arreglo está ordenado o casi ordenado. Si por ejemplo k siempre se halla en su posición correcta, el proceso de partición de un segmento de n elementos crea dos subarchivos de tamaño 0 y n-1. El resultado es que se requieren n procesos de partición, el primero de tamaño n, el segundo de n-1, el tercero de n-2 y así sucesivamente. El número total de comparaciones para ordenar todo el archivo sería

$$n + (n-1) + (n-2) + \dots + 2$$

lo cual equivale a $O(n^2)$.

```
PROGRAM Quicksort;
( Este programa  ordena un arreglo de n numeros por el metodo de
 intercambio con particion; presentamos la implementacion de ma-
 nera recursiva)

CONST
   n=10;
TYPE
   archivo=ARRAY[1..n] OF INTEGER;
   apunt=1..n;
VAR
   ent:TEXT;  (archivo de numeros a ordenar)
   x:archivo; (guarda los elementos a ordenar)

   {**************************************************************}
   PROCEDURE Leenumeros;
   VAR  i:INTEGER;
   BEGIN
      WRITELN('Numeros a ordenar:');
      ASSIGN(ent,'DATOS.DAT');
      RESET(ent);
      FOR i:=1 TO  n  DO
         BEGIN
           READ(ent,x[i]); WRITE(x[i],'   ');
        END; WRITELN;
   END;
   {**************************************************************}
   PROCEDURE Escribeproceso(m,n: INTEGER);
   VAR
      i:INTEGER;
   BEGIN
      IF ((m=1) AND (n=10)) THEN
      BEGIN
        WRITELN;
        WRITE('La tabla siguiente da el estado del arreglo en ');
        WRITELN('cada'):
        WRITELN('llamada a Quick y se muestran las particiones');
        FOR i:=1 TO 10 DO
           WRITE('x[',i,']','   ');
        WRITE('m     n');  WRITELN;
      END;
      FOR i:=1 TO 10 DO
         IF i=m THEN WRITE(' ','[',x[i],'  ')
         ELSE
            IF i=n THEN WRITE('   ',x[i],']',' ')
            ELSE
               WRITE('   ',x[i],'  ');
      WRITE(' ',m,'    ',n);  WRITELN;
   END;
   {**************************************************************}
   PROCEDURE Escribesolucion;
   VAR
      i:INTEGER;
   BEGIN
      WRITELN; WRITELN('Numeros ordenados en forma ascendente:');
      FOR i:=1 TO 10 DO
```

```
        WRITE(x[i], '  ');
      WRITELN;
    END;
    {**************************************************************}
    PROCEDURE Reorganiza(m,n:INTEGER; VAR j:INTEGER);
    {reorganiza los elementos del subarreglo con limites m y n,
     tal que x[m] estara en x[j] (j es un parametro de salida) y:
                   x[i] <= x[j] para m <= i < j
                   x[i] >= x[j] para j < i <= n
     x[m] estara ahora en su posicion final}
     VAR
      down,up:apunt;
      k:INTEGER;
    BEGIN
      k:=x[m];
      j:=m;
      up:=n;
      down:=m;
      REPEAT
        WHILE (up > down) AND (x[up] >=k) {se recorre hacia abajo}
            DO up:=up-1;
        IF up <> down THEN
            BEGIN
               x[down]:=x[up]; {se recorre hacia arriba el arreglo}
               WHILE (down < up) AND (x[down] <= k)
                  DO down:=down+1;
               IF down <> up THEN
                  x[up]:=x[down]
            END
      UNTIL down=up;
      j:=down;
      x[j]:=k
    END;
    {**************************************************************}
    PROCEDURE Quick(m,n:INTEGER);
    { Se define como un proceso recursivo}
    VAR
       j:INTEGER;
    BEGIN
       IF m < n THEN
          BEGIN  Escribeproceso(m,n);
             Reorganiza(m, n, j);
             Quick(m, j-1);{ordena subarreglo entre x[m] y x[j-1]}
             Quick(j+1, n);{ordena subarreglo entre x[j+1] y x[n]}
          END;
    END;

{*******************************************************************}
                       { Programa Principal }
BEGIN
   Leenumeros;
   Quick(1,n);
   Escribesolucion;
END.
```

Numeros a ordenar:
26 54 37 11 61 17 59 15 48 19

La tabla siguiente da el estado del arreglo en cada
llamada a Quick y se muestran las particiones

x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]	m	n
[26	54	37	11	61	17	59	15	48	19]	1	10
[19	15	17	11]	26	61	59	37	48	54	1	4
[11	15	17]	19	26	61	59	37	48	54	1	3
11	[15	17]	19	26	61	59	37	48	54	2	3
11	15	17	19	26	[61	59	37	48	54]	6	10
11	15	17	19	26	[54	59	37	48]	61	6	9
11	15	17	19	26	[48	37]	54	59	61	6	7

Numeros ordenados en forma ascendente:
11 15 17 19 26 37 48 54 59 61

ORDENAMIENTO DE GRUPO (Heapsort)

El algoritmo Heapsort posee la ventaja de mantener tiempos similares para la mayoría de los conjuntos por ordenar, sin importar la distribución inicial de éstos, excepto para aquellos conjuntos semiordenados, en los cuales es muy recomendable su uso.

Se define un **ordenamiento de grupo de tamaño n o estructura heap** como un árbol binario casi completo de n nodos (al que pueden faltarle algunas de las hojas situadas más a la derecha en el último nivel) tal que el contenido de cada nodo es menor o igual al de su padre. La raíz del árbol heap es el de valor más grande.

ALGORITMO

El Heapsort se puede ver como un método de dos etapas:

- Los n números de entrada representados en un árbol binario casi completo (con implementación de arreglo) son convertidos en estructura heap.
  Una manera de construir la estructura heap es la siguiente:
    Se posiciona en la mitad del arreglo (posición i).
    El siguiente procedimiento se repite hasta que i=1.
      Se obtiene el hijo mayor de las posiciones 2i y 2i+1.
        El hijo mayor se compara con el padre, si éste resulta menor se intercambian.
        Se decrementa i en 1.
- Los números ordenados se obtienen sacando elementos sucesivos de la raíz del árbol hasta que queda vacía. Después de extraer cada raíz se actualiza el árbol restante en heap.

ANALISIS

El ordenamiento Heapsort no es muy eficiente para valores de n pequeños. El orden depende de la profundidad o nivel del árbol binario. Un árbol binario balanceado (porque es casi completo) con n nodos tiene un número máximo de niveles igual a $\log_2 n$, por lo tanto el número de comparaciones en cada actualización del heap es de a lo más $\log_2 n$. El heap se actualiza n-1 veces por lo que el número total de comparaciones es del orden O(n log n).

```
PROGRAM Heapsort;
( Ordena n numeros por el metodo de ordenamiento de grupo.
  Presentamos la implementacion en un arreglo, el cual representa
  un arbol binario  casi corpleto, tal que el  nodo i es el padre
  de los nodos 2i y 2i+1 en el arreglo)
CONST
   n=10;
TYPE
   arbol=array[1..n] OF INTEGER;
VAR
   ent:TEXT; (archivo de numeros a ordenar)
   x:arbol;  (estructura de almacenamiento de los numeros)

   ( El siguiente procedimiento aparece en el programa anterior)
   PROCEDURE Leenumeros;

   (***********************************************************)
   PROCEDURE Escriberesultado;
   VAR i:INTEGER;
   BEGIN
      FOR i:=1 TO n DO
         WRITE(x[i],'  ');
      WRITELN;
   END;
   (***********************************************************)
   PROCEDURE Intercambia(VAR x,y:INTEGER);
   (Intercambia los elementos x e y)
   VAR t:INTEGER;
   BEGIN
      t:=x;  x:=y;  y:=t;
   END;
   (***********************************************************)
   PROCEDURE Acomoda(VAR x:arbol; i,n:INTEGER);
   ( Accomoda un arbol binario con raiz i para satisfacer la pro-
     piedad heap)
   VAR
      j,k:INTEGER;
      arreglado:BOOLEAN;
   BEGIN
      arreglado:=FALSE;
      k:=x[i];
      j:=2*i;
      WHILE ((j <= n) AND NOT arreglado) DO
         BEGIN (primero encontramos el mayor de los hijos)
            IF j < n THEN
               IF x[j] < x[j+1] THEN j:=j+1;
           (comparamos el hijo mayor con k, si k es mayor ya esta
            arreglado)
            IF k >= x[j] THEN
               arreglado:=TRUE
            ELSE
              BEGIN
                x[j DIV 2]:=x[j];(movemos el nodo j hacia arriba)
                 j:=j*2;
              END;
         END;
```

```
        x[j DIV 2]:=k;
        IF ((i=1) AND (n=10)) THEN
           BEGIN
              WRITELN;  WRITELN('Numeros en la estructura heap:');
           END;
        IF ((i=1) AND (n=9))  THEN
           BEGIN
              WRITELN;  WRITELN('Proceso de ordenamiento');
           END;
     END;
     {*************************************************************}
     PROCEDURE Hsort(VAR x:arbol; n:INTEGER);
     { El archivo x es ordenado en forma ascendente en el mismo
      arreglo}
     VAR i:INTEGER;
     BEGIN
        FOR i:=(n DIV 2) DOWNTO 1 DO    {se convierte x en un heap}
           Acomoda(x,i,n); Escriberesultado;
        FOR i:=(n-1) DOWNTO 1 DO        {ordena x}
           BEGIN
              Intercambia(x[1],x[i+1]);
              Acomoda(x,1,i);           {actualiza heap}
              Escriberesultado;
           END; WRITELN;
     END;

{*************************************************************}
                    { Progama Principal }
BEGIN
   Leenumeros;
   Hsort(x,n);
   Escriberesultado;
END.
```

```
Numeros a ordenar:
26  54  37  11  61  17  59  15  48  19

Numeros en la estructura heap:
61 54 59 48 26 17 37 15 11 19

Proceso de ordenamiento
59 54 37 48 26 17 19 15 11 61
54 48 37 15 26 17 19 11 59 61
48 26 37 15 11 17 19 54 59 61
37 26 19 15 11 17 48 54 59 61
26 17 19 15 11 37 48 54 59 61
19 17 11 15 26 37 48 54 59 61
17 15 11 19 26 37 48 54 59 61
15 11 17 19 26 37 48 54 59 61
11 15 17 19 26 37 48 54 59 61

11 15 17 19 26 37 48 54 59 61
```

```
      x[j DIV 2]:=k;
      IF ((i=1) AND (n=10)) THEN
         BEGIN
            WRITELN;  WRITELN('Numeros en la estructura heap:');
         END;
      IF ((i=1) AND (n=9))  THEN
         BEGIN
            WRITELN;  WRITELN('Proceso de ordenamiento');
         END;
   END;
   {************************************************************}
   PROCEDURE Hsort(VAR x:arbol; n:INTEGER);
   { El archivo x es ordenado en forma ascendente en el mismo
    arreglo}
   VAR i:INTEGER;
   BEGIN
      FOR i:=(n DIV 2) DOWNTO 1 DO    {se convierte x en un heap}
         Acomoda(x,i,n); Escriberesultado;
      FOR i:=(n-1) DOWNTO 1 DO        {ordena x}
         BEGIN
            Intercambia(x[1],x[i+1]);
            Acomoda(x,1,i);           {actualiza heap}
            Escriberesultado;
         END; WRITELN;
   END;

{***************************************************************}
                    { Progama Principal }
BEGIN
   Leenumeros;
   Hsort(x,n);
   Escriberesultado;
END.
```

```
Numeros a ordenar:
26  54  37  11  61  17  59  15  48  19

Numeros en la estructura heap:
61 54 59 48 26 17 37 15 11 19

Proceso de ordenamiento
59 54 37 48 26 17 19 15 11 61
54 48 37 15 26 17 19 11 59 61
48 26 37 15 11 17 19 54 59 61
37 26 19 15 11 17 48 54 59 61
26 17 19 15 11 37 48 54 59 61
19 17 11 15 26 37 48 54 59 61
17 15 11 19 26 37 48 54 59 61
15 11 17 19 26 37 48 54 59 61
11 15 17 19 26 37 48 54 59 61

11 15 17 19 26 37 48 54 59 61
```

ORDENAMIENTO DE SHELL

Este método de clasificación (ordenamiento de disminución incremental) lleva el nombre de su autor: D. L. Shell. Ordena subarchivos separados del archivo original. Estos subarchivos contienen elementos del archivo original separados por k unidades. El valor de k se denomina incremento.

ALGORITMO

- Se calcula un número k en términos de la longitud n del arreglo x, el cual determina un cierto intervalo entre los elementos que se van a comparar. Por ejemplo: k=n/2.
- Se inicia el ordenamiento (utilizando inserción simple) de los subarchivos, cuyos elementos están separados k unidades del arreglo original.
- Una vez que se recorrió todo el arreglo, se escoge un nuevo valor de k (más pequeño) bajo el criterio k=k/2 y se regresa al paso anterior. Este proceso se repite hasta que k=1, en cuyo caso serà el último recorrido del arreglo.

Mostramos los subarchivos a ordenar con cada incremento (k), en un archivo de 10 elementos:

Primera iteración (incremento=5)
(x[1],x[6])
(x[2],x[7])
(x[3],x[8])
(x[4],x[9])
(x[5],x[10])
Segunda iteración (incremento=2)
(x[1],x[3],x[5],x[7],x[9])
(x[2],x[4],x[6],x[8],x[10])
Tercera iteración (incremento=1)
(x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10])

Las iteraciones sucesivas entremezclan los subarchivos de tal manera que el archivo en forma total está casi ordenado cuando incremento es igual a 1 en la última iteración.

ANALISIS

El algoritmo de Shell es recomendable sólo para subconjuntos de pequeña magnitud, en particular para aquellos de magnitud menor o igual a 50 elementos. La distribución inicial de los elementos por ordenar no influye en el comportamiento de dicho algoritmo, éste depende de la secuencia de incrementos usada. Es uno de los métodos más difíciles de determinar el orden, aproximadamente el tiempo requerido en el mejor de los casos es $O(n^{1.2})$. El número de comparaciones se reduce cuando la secuencia es de primos relativos.

```
PROGRAM Shell;
{  Este programa ordena un arreglo de n numeros por el metodo de
 Shell (ordenamiento de disminucion incremental)}
CONST
   n=10;
TYPE
   archivo=array[1..n] OF INTEGER;
VAR
   ent:TEXT; {archivo de numeros a ordenar}
   x:archivo:{estructura de almacenamiento de los numeros}

   { El siguiente procedimiento aparece anteriormente}
   PROCEDURE Leenumeros;

   {*****************************************************************}
   PROCEDURE Escriteresultado(inc:INTEGER);
   VAR
      i:INTEGER;
   BEGIN
      WRITE('Incremento=',inc,'     ');
      FOR i:=1 TO n DO
         WRITE(x[i],'   ');
      WRITELN;
   END;
   {*****************************************************************}
   PROCEDURE Shell(VAR x:archivo; n:INTEGER);
   VAR
     inc:INTEGER;  {indica tamano del incremento}
     j,i,y:INTEGER;
     arreglado:BOOLEAN;
   BEGIN
      inc:=n DIV 2;
      WHILE inc >= 1 DO
         BEGIN
            FOR j:=inc+1 TO n DO
               BEGIN
                  { se inserta el elemerto x[j] en su posicion
                  apropiada en el archivo }
                  y:=x[j];
                  i:=j-inc;
                  arreglado:=FALSE;
                  WHILE (i >=1) AND (NOT arreglado) DO
                     IF y < x[i] THEN
                        BEGIN
                           x[i+inc]:=x[i];
                           i:=i-inc;
                        END
                     ELSE
                        arreglado:=TRUE;
                  x[i+inc]:=y;
               END;
            Escriberesultado(inc);
            inc:=inc DIV 2;  {se decrementa el incremento}
         END;
   END;
{*****************************************************************}
```

(Programa Principal)

```
BEGIN
   Leenumeros;
   Shell(x,n);
END.
```

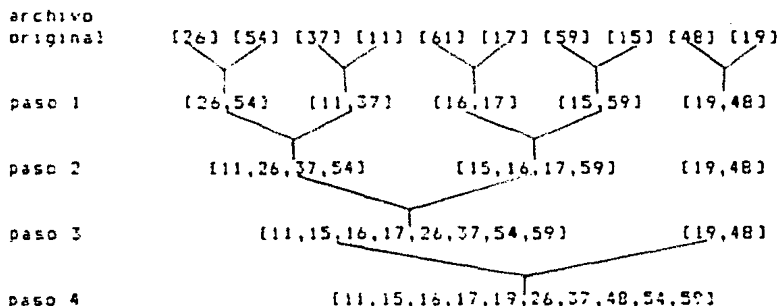
```
Numeros a ordenar:
26  54  37  11  61  17  59  15  48  19

Incremento=5   17  54  15  11  19  26  59  37  48  61
Incremento=2   15  11  17  26  19  37  48  54  59  61
Incremento=1   11  15  17  19  26  37  48  54  59  61
```

ORDENAMIENTO DE CONCATENACION (Merge)

El proceso de concatenación consiste en la combinación o mezcla de dos o más archivos ordenados, con lo que se forma un tercer archivo ordenado.

ALGORITMO

Inicialmente se consideran los n datos del archivo a ordenar como n archivos ordenados de tamaño uno. Estos últimos son concatenados por parejas adyacentes. En este caso, entonces, obtenemos aproximadamente n/2 archivos de tamaño dos . Estos n/2 archivos son concatenados nuevamente por parejas y el proceso se repite hasta que queda únicamente un archivo de tamaño n, como se muestra en la siguiente figura:



El algoritmo es implementado en forma no recursiva. Cuenta con el procedimiento Merge, que concatena dos archivos de manera ordenada.

ANALISIS

El ordenamiento Merge es del orden O(n log n), debido a que existen aproximadamente $\log_2 n$ pasos (como se puede ver en la figura de arriba) y en cada paso se hacen n comparaciones.

Sin embargo con respecto a los otros ordenamientos el Merge requiere de un arreglo auxiliar durante el proceso.

```
PROGRAM Mergesort;
{ Este programa  ordena un archivo de n numeros por el metodo de
 concatenacion (Merge). Se trata de ir combinando  ordenadamente
 subarchivos adyacentes de longitudes l, cada vez menores, hasta
 llegar a l=1}

CONST
   n=10;
TYPE
   archivo=ARRAY[1..n] OF INTEGER;
VAR
   ent:TEXT;  {archivo de numeros a ordenar}
   x:archivo; {estructura de almacenamiento de los numeros}

   { El siguiente procedimiento aparece anteriormente}
   PROCEDURE Leenumeros;

   {***********************************************************}
   PROCEDURE EscribeResultado(x:archivo; l:INTEGER);
   VAR
      i:INTEGER;
   BEGIN
      IF l=1 THEN
         BEGIN
            WRITE('Se muestran el archivo en cada paso de ');
            WRITELN('concatenacion');
            WRITELN('y la longitud l de los subarchivos');
            WRITELN;
            FOR i:=1 TO n DO
               WRITE('x[',i,']',' ');
            WRITELN(' ','long');
         END;
      IF l > n THEN WRITELN;
      FOR i:=1 TO n DO
         WRITE(' ',x[i],'  ');
      IF l <= n THEN
         WRITELN('    ',l)
   END;
   {***********************************************************}
   PROCEDURE Merge(VAR x,z:archivo; p,q,r:INTEGER);
   {(x[p],...,x[q]) y (x[q+1],...,x[r]) son dos listas ordenadas,
    tal  que  x[p] <=,...<= x[q] y x[q+1] <=,...,<= x[r].  Estos
    elementos son combinados para obtener la lista ordenada
    (z[p],...,z[r]), tal que z[p] <=, ... , <= z[r]}
    VAR
      i,j,k,t:INTEGER;
   BEGIN
      i:=p;
      k:=p;
      j:=q+1;  {i,j,k son posiciones en los tres archivos}
      WHILE ((i <= q) AND (j <= r)) DO
         BEGIN
            IF x[i] <= x[j] THEN
               BEGIN
                  z[k]:=x[i]; i:=i+1;
               END
```

```
          ELSE
             BEGIN
                z[k]:=x[j]; j:=j+1;
             END;
          k:=k+1;
       END;
    IF i > q THEN              {(z[k],...,z[q])=(x[j],...,x[r])}
       FOR t:=j TO r DO
          z[k+t-j]:=x[t]
    ELSE                       {(z[k],...,z[q])=(x[j],...,x[q])}
       FOR t:=i TO  q DO
          z[k+t-i]:=x[t];
END;
{**************************************************************}
PROCEDURE Mpass(VAR x,y:archivo; n,l:INTEGER);
{ Este algoritmo se encarga de llamar al procedimiento Merge
 para ejecutar  las concatenaciones  necesarias; l indica la
 longitud de los subarchivos a concatenar del arreglo x a y,
 n es el numero de elementos del archivo x}
VAR
   i,t:INTEGER;
BEGIN
   i:=1;
   WHILE i <= (n-2*l+1) DO
      BEGIN
         {se llama a Merge con los limites de cada subarchivo}
         Merge(x, y, i, i+l-1, i+2*l-1);
         i:=i+2*l;
      END ;
   {se concatena el archivo restante de longitud < l}
   IF (i+l-1) < n THEN
      Merge(x, y, i, i+l-1, n)
   ELSE
      FOR t:=i TO n DO
         y[t]:=x[t];
END;
{**************************************************************}
PROCEDURE Msort(VAR x:archivo; n:INTEGER);
{ Ordena los elementos del archivo x=(x[1],...,x[n])}
VAR
   l:INTEGER;
   y:archivo;
BEGIN
   {l es la medida del subarchivo actual a ser combinado}
   l:=1;
   WHILE l < n DO
      BEGIN
         Mpass(x, y, n, l);  EscribeResultado(y,l);
         l:=l*2;
         {se intercambian los papeles de x y y}
         Mpass(y, x, n, l);  EscribeResultado(x,l);
         l:=2*l;
      END;
      EscribeResultado(x,l);
END;
{**************************************************************}
```

```
                    { Programa Principal }
BEGIN
   Leenumeros;
   Msort(x,n);
END.
```

```
Numeros a ordenar:
26  54  37  11  61  17  59  15  48  19

Se muestran el archivo en cada paso de concatenacion
y la longitud l de los subarchivos

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8] x[9] x[10]  long
 26   54   11   37   17   61   15   59   19   48      1
 11   26   37   54   15   17   59   61   19   48      2
 11   15   17   26   37   54   59   61   19   48      4
 11   15   17   19   26   37   48   54   59   61      8

 11   15   17   19   26   37   48   54   59   61
```

BUSQUEDA

Ademas de clasificar un archivo, es necesario contar con algoritmos que nos permitan encontrar a los elementos (registros) para cualquier operación que se pretenda.

El proceso de búsqueda en un archivo depende directamente de la organización (estructura) dèl archivo. Este puede ser organizado como un arreglo de registros, una lista encadenada, un árbol, o hasta una gráfica. Ya que diferentes técnicas de búsqueda pueden ser apropiadas para distintas organizaciones, el archivo generalmente está diseñado para alguna técnica de búsqueda específica.

DEFINICIONES BASICAS

Asociada con cada registro del archivo existe una llave que es usada para accesar los diferentes registros. La asociación entre un registro y su llave puede ser muy simple o compleja. En el caso simple, la llave está contenida dentro del registro en algún sitio específico. Cuando esto sucede la llave se denomina **interna.** En otras situaciones hay una tabla separada de llaves, la cual incluye los apuntadores a los registros. Estas llaves se llaman **externas.**

Un algoritmo de búsqueda acepta un argumento **a** y trata de encontrar un registro cuya llave es **a**.

El archivo puede estar contenido en forma completa en memoria principal, en un almacenamiento auxiliar o partido entre los dos.

De acuerdo a las distintas condiciones que se presenten, se requieren diferentes técnicas de búsqueda.

Aquellas búsquedas en las cuales el archivo está completamente contenido en la memoria, son llamadas **búsquedas internas,** mientras que aquellas en las cuales casi todo el archivo es mantenido en almacenamiento auxiliar son denominadas **búsquedas externas.** Al igual que en el caso de ordenamiento, trataremos solamente lo que se refiere a búsquedas internas.

Presentamos los siguientes algoritmos de búsqueda:

1. Técnicas básicas de búsqueda
   - Búsqueda secuencial
   - Búsqueda secuencial indexada
   - Búqueda binaria
2. Búsqueda en árbol
   - Arboles balanceados
3. Función de dispersión (Hash)

BUSQUEDA SECUENCIAL

Este es el método más simple de búsqueda que existe. Es aplicable a un archivo que está organizado ya sea en un arreglo o como una lista encadenada.

ALGORITMO
Se analizan los elementos uno tras otro hasta encontrar el elemento deseado.

Asumamos que x es un arreglo de n registros y que los registros contienen únicamente las llaves k (caso simple).
Escribiremos el algoritmo de búsqueda secuencial.

```
FUNCTION Busca(k:inftipo; x:archivo):INTEGER;
(Busca la llave k en el archivo x organizado como un arreglo y
 retorna la posición que ocupa, en caso de no encontrarlo
 retorna el valor de 0)
VAR
   encontrado:BOOLEAN;
   i:INTEGER;
BEGIN
   encontrado:=FALSE;
   i:=1;
   WHILE ((i<=n) AND (NOT encontrado)) DO
      IF k=x[i] THEN
         BEGIN
            Busca:=i; encontrado:=TRUE;
         END
      ELSE  i:=i+1;
   IF NOT encontrado THEN
      Busca:=0;
END;
```

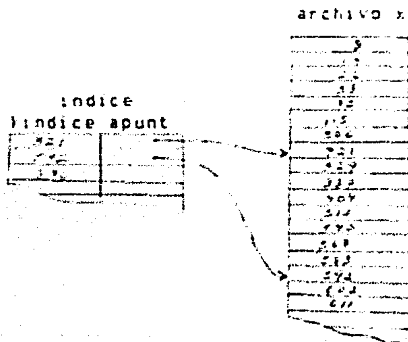
El número de comparaciones para la búsqueda de un elemento en un arreglo (tabla) depende de dónde se encuentre dicho elemento.
Si el registro es el primero en la tabla, entonces se realiza una sola comparación; si el registro es el último en la tabla, entonces se requieren n comparaciones. Si es probable que el argumento aparezca en cualquier posición en la tabla, entonces la búsqueda secuencial tomará en promedio n/2 comparaciones y una búsqueda sin éxito tomará n comparaciones. En cualquier caso el número de comparaciones es del orden O(n).
Si los registros que son accesados con frecuencia se colocan al comienzo del archivo, el número promedio de comparaciones se reduce considerablemente debido a que los registros que son accesados más frecuentemente requieren la menor cantidad de tiempo para ser encontrados.

Hemos tratado únicamente búsquedas pero generalmente se puede dar el caso en que la llave no exista y se desea insertar. En este caso se trata de búsqueda e inserción o también se puede requerir borrar una llave, en tal caso se trata de búsqueda y borrado. Estas operaciones de inserción y borrado se pueden realizar de varias maneras.

BUSQUEDA SECUENCIAL INDEXADA

En el caso de que el archivo esté almacenado en orden ascendente o descendente con respecto a las llaves de los registros, existen varias técnicas que se pueden utilizar para mejorar la eficiencia de búsqueda. Una de ellas es este método.

Se requiere una tabla auxiliar denominada índice, adicional al archivo ordenado. Cada elemento de la tabla índice consta de una llave kindice y un apuntador al registro en el archivo que corresponda a kindice. Los elementos en el índice, al igual que los elementos en el archivo, deben estar ordenados en base a la llave. La tabla de índice es de tamaño mucho menor que el archivo. Si la tabla índice es 1/8 del tamaño del archivo, entonces cada ocho registros del archivo están representados en la tabla índice.



Un Archivo Secuencial Indexado

ALGORITMO

Para buscar una llave k en el archivo x de tamaño n se procede de la siguiente manera:

Se realiza una búsqueda secuencial en la tabla de índices (que es mas pequeña) en lugar de hacerlo en la tabla x. Una vez que se ha encontrado un índice correcto, se efectúa una segunda búsqueda secuencial únicamente en la parte reducida (con límites l y u) de la tabla x.

Consideramos que kindice es un arreglo de llaves en la tabla índice y apunt un arreglo de apuntadores a los registros actuales del archivo. Maxind es el tamaño de la tabla de índice.

```
FUNCTION Busca(k:inftipo; indice:tabla; x:archivo):INTEGER;
{Busca el elemento k en el archivo x (ordenado en forma
 ascendente por alguno de los metodos de ordenamiento ya
 tratados) y retorna la posicion que ocupa dentro del archivo,
 en caso de no encontrarlo retorna el valor 0}
VAR
   encontrado:BOOLEAN;
   i,l,u:INTEGER;
```

```
BEGIN
   encontrado:=FALSE;
   i:=1;
   WHILE ((i <= maxind) AND (NOT encontrado)) DO
      IF kindice(i) > k THEN
         encontrado:=TRUE
      ELSE
         i:=i+1;
   IF i=1 THEN
      l:=1
   ELSE
      l:=apunt(i-1);
   IF encontrado THEN
      u:=apunt(i)-1
   ELSE
      u:=n;
   {se busca en la tabla entre posiciones l y u}
   j:=l;
   encontrado:=FALSE;
   WHILE ((j <= u) AND (NOT encontrado)) DO
      IF x(j)=k THEN
         encontrado:=TRUE
      ELSE
         j:=j+1;
   IF encontrado THEN
      Busca:=j
   ELSE
      Busca:=0;
END;
```

La ventaja del metodo es que se reduce considerablemente el tiempo de búsqueda, debido a que la búsqueda se realiza primero en la tabla de índices, después en una parte reducida del archivo. Estas búsquedas se llevan a cabo de manera secuencial o por otros métodos, como la búsqueda binaria, la cual describiremos en seguida.

Esta forma de organización de archivo utilizando una tabla de índice es aplicable a una tabla ordenada almacenada en forma de arreglo (como se ha tratado aquí) al igual que en una almacenada como una lista encadenada. La utilización de una lista encadenada implica un espacio mayor para apuntadores pero las inserciones y eliminaciones pueden ser realizadas más rapidamente.

BUSQUEDA BINARIA

Es uno de los métodos más eficientes de búsqueda en archivos ordenados, sin la utilización de índices auxiliares.

ALGORITMO

Compara el elemento k a buscar con la llave de la mitad del arreglo (x[mid]), de manera que si k es menor que x[mid] busca en la parte superior y si k es mayor que x[mid] en la inferior. Se hace uso de dos apuntadores: l y u.

```
FUNCTION Busca(k:inftipo; x:archivo):INTEGER;
(Esta función busca el elemento k en el archivo x ordenado
 y retorna la posición que ocupa dentro del arreglo, en caso
 de no encontrarlo retorna el valor 0)
VAR
   mid,l,u:INTEGER;
   encontrado:BOOLEAN;
BEGIN
   l:=1;  u:=n;
   encontrado:=FALSE;
   WHILE ((l<=u) AND (NOT encontrado)) DO
      BEGIN
         mid:=(l+u)DIV 2;
         IF k=x[mid] THEN
            encontrado:=TRUE
         ELSE
            IF k<x[mid] THEN  u:=mid-1
            ELSE  l:=mid+1
      END;
   IF encontrado THEN
      Busca:=mid
   ELSE
      Busca:=0
END;
```

Este algoritmo de búsqueda binaria es el mismo para búsqueda en árboles binarios, debido a que el arreglo ordenado puede verse como un árbol de búsqueda binaria. El elemento del centro del arreglo se puede considerar como la raíz del árbol, donde la mitad inferior del arreglo (todos aquellos elementos que son menores que el de en medio) es el subárbol izquierdo y la mitad superior (todos aquellos elementos que son mayores que el de en medio) es el subárbol derecho.

Con respecto a tiempo de búsqueda, este algoritmo es del orden O(log n).

La búsqueda binaria desafortunadamente sólo se puede utilizar en tablas que han sido almacenadas como un arreglo, debido a que los índices de los elementos del arreglo se consideran como enteros consecutivos.

BUSQUEDA EN ARBOL BINARIO BALANCEADO

En el tema de árbol hemos presentado una aplicación de árbol binario como árbol de búsqueda, donde la inserción, eliminación o búsqueda se realizan de manera eficiente.

El tiempo para una búsqueda en un árbol de búsqueda binaria varía según la estructura del árbol. Si insertamos elementos, la estructura del árbol de búsqueda resultante depende del orden en el cual se insertan los registros. Si los registros son insertados en orden (o en sentido inverso), el árbol resultante contendrá todos los enlaces izquierdos nulos (o derechos) tal que la búsqueda en el árbol se reduce a una búsqueda secuencial. Sin embargo, si la mitad de los registros insertados después de cualquier registro con llave k tiene llaves menores que k y la mitad tiene llaves mayores que k, se obtiene un **árbol balanceado**, en el cual se hace bastante eficiente la localización de un elemento.

DEFINICIONES PREVIAS

Altura de un árbol es el nivel máximo de sus hojas (esto es también conocido como profundidad del árbol).

Equilibrio o balance de un nodo en un árbol binario se define como la altura del subárbol derecho menos la altura del subárbol izquierdo correspondiente.

DEFINICION

Un **árbol binario balanceado** (llamado también árbol AVL en honor de sus inventores: Adelson-Velskii y Landis) es aquel en el que las alturas de los dos subárboles para cada nodo difieren a lo más en 1.

ALGORITMO DE INSERCION

Algoritmo recursivo. Al insertar un nuevo nodo en un árbol balanceado dada una raíz r con subárboles I y D, puede desbalancear al árbol o mejorar su equilibrio igualando las alturas. La eficiencia de un algoritmo que inserte y rebalancee dependerá de la forma en que se almacene la información relativa al balance del árbol. En nuestro caso consiste en atribuir a, y almacenar con, cada nodo un factor de equilibrio explícito. El factor de equilibrio puede ser -1, 0 ó 1, de acuerdo con la definición dada anteriormente.

La definición de un nodo del árbol se amplía entonces a:

```
TYPE
   nodo=RECORD
          llave:INTEGER;
          contador:INTEGER;
          izquierdo:apuntador;
          derecho:apuntador;
          equi:-1..1
        END;
```

El proceso de inserción de un nodo consta fundamentalmente de las tres partes consecutivas siguientes:

1. Seguir el camino de búsqueda hasta que se comprueba que la llave aún no está en el árbol.

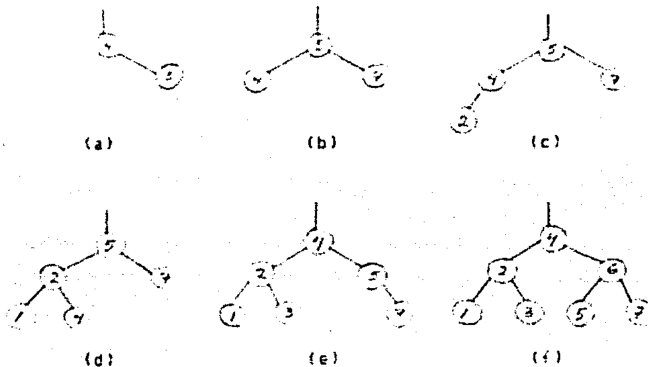
2. Insertar el nuevo nodo y determinar el factor de equilibrio resultante.
3. Volver, siguiendo el camino de búsqueda, y comprobar el factor de equilibrio de cada nodo.

Debido a la recursividad del algoritmo, éste alberga una operación adicional en el camino de vuelta atrás a lo largo del camino de búsqueda. En cada paso se debe mandar información sobre si la altura del subárbol (en el cual se ha realizado la inserción) ha aumentado o no. Por lo tanto se extiende la lista de parámetros del procedimiento con el boolean h, que significa que la altura del subárbol se ha incrementado.

Las operaciones necesarias para rebalancear el equilibrio se expresan en su totalidad como secuencias de reasignaciones de apuntadores. De hecho, los apuntadores se intercanbian cíclicamente, lo que resulta una rotación simple o doble de los dos o tres nodos en cuestión. Además de la rotación de los apuntadores, se deben también ajustar los respectivos factores de equilibrio de los nodos.

En la siguiente figura se da un ejemplo de inserción en un árbol equilibrado, con el fin de mostrar las rotaciones simples y dobles.

Considere el árbol binario (a) que está formado por dos únicos nodos. La inserción de la clave 7 produce primero un árbol desequilibrado. Se equilibra con una rotación DD simple, resultando el árbol perfectamente equilibrado (b). A continuación, una inserción de los nodos 2 y 1 provoca un desequilibrio del subárbol con raíz 4. Este subárbol se equilibra con una rotación [illegible] simple (d). La inserción subsiguiente de la clave 3 anula inmediatamente el equilibrio del nodo raíz 5. El equilibrio se restaura a continuación mediante la rotación ID doble, que es más complicada; el resultado es el árbol (e). El único nodo que puede ahora perder el equilibrio con una inserción es el 5. De hecho, la inserción del nodo 6 debe utilizar el último caso de equilibrio, la rotación DI doble. El árbol final resultante se muestra en la figura (f).



Inserciones en un árbol equilibrado

Presentamos el siguiente algoritmo de inserción.

```
PROCEDURE Buscar(x:INTEGER; VAR p:apunt; VAR h:BOOLEAN);
{ Busca alguna llave x en el árbol  apuntado por p, si no se en-
cuentra lo inserta y se encarga de mantener balanceado el árbol}
VAR      {h=FALSE}
   p1,p2:apunt;
BEGIN
   IF p=NIL THEN
      BEGIN     {la llave no está en el árbol; insertarla}
         NEW(p);  h:=TRUE;
         WITH p^ DO
            BEGIN
               llave:=x;  contador:=1;
               izquierdo:=NIL;  derecho:=NIL;  equi:=0;
            END;
      END
   ELSE
   IF x < p^.llave THEN
      BEGIN
         Buscar(x,p^.izquierdo,h);
         IF h THEN  {la rama izquierda ha crecido}
         CASE p^.equi OF
           1:  BEGIN
                 p^.equi:=0;  h:=FALSE;
               END;
           0:  p^.equi:=-1;
          -1:  BEGIN  {reequilibrar}
                 p1:=p^.izquierdo;
                 IF p1^.equi=-1 THEN
                   BEGIN  {rotación II simple}
                      p^.izquierdo:=p1^.derecho; p1^.derecho:=p;
                      p^.equi:=0;  p:=p1;
                   END
                 ELSE
                   BEGIN  {rotación ID doble}
                    p2:=p1^.derecho;
                    p1^.derecho:=p2^.izquierdo; p2^.izquierdo:=p1;
                    p^.izquierdo:=p2^.derecho; p2^.derecho:=p;
                    IF p2^.equi=-1 THEN p^.equi:=1
                    ELSE p^.equi:=0;
                    IF p2^.equi=1 THEN p1^.equi:=-1
                    ELSE p1^.equi:=0;
                    p:=p2;
                  END;
                 p^.equi:=0; h:=FALSE;
               END;
         END;
      END
   ELSE
   IF x > p^.llave THEN
      BEGIN
         Buscar(x,p^.derecho,h);
         IF h THEN  {la rama derecha ha crecido}
         CASE p^.equi OF
```

```
              -1:  BEGIN
                     p^.equi:=0;  h:=FALSE;
                   END;
               0:  p^.equi:=1;
               1:  BEGIN  (reequilibrar)
                     p1:=p^.derecho;
                     IF p1^.equi=1 THEN
                       BEGIN  (rotación DD simple)
                         p^.derecho:=p1^.izquierdo; p1^.izquierdo:=p;
                         p^.equi:=0;  p:=p1;
                       END
                     ELSE
                       BEGIN  (rotación DI doble)
                         p2:=p1^.izquierdo;
                         p1^.izquierdo:=p2^.derecho; p2^.derecho:=p1;
                         p^.derecho:=p2^.izquierdo; p2^.izquierdo:=p;
                         IF p2^.equi=1 THEN p^.equi:=-1
                         ELSE p^.equi:=0;
                         IF p2^.equi=-1 THEN p1^.equi:=1
                         ELSE p1^.equi:=0;
                         p:=p2;
                       END;
                       p^.equi:=0;  h:=FALSE;
                   END;
          END;
      END
    ELSE
      BEGIN
         p^.contador:=p^.contador+1;  h:=FALSE;
      END;
END;
```

ALGORITMO DE BORRADO

Dado un árbol equilibrado se borrará alguna llave. El borrado es un poco más difícil que la inserción. La operación de reequilibrado es esencialmente la misma que la de inserción.

Los casos que se pueden presentar son los siguientes:

- -Borrar nodos terminales: se quita y se reequilibra.
- -Borrar nodos con un descendiente único: se quita y el descendiente pasa a ocupar el lugar y se reequilibra.
- -Borrar nodos que tienen dos subarboles: se quita y se sustituye por el nodo situado más a la derecha del subárbol izquierdo.

Tal como se hizo en el caso de inserción, se añade un parámetro booleano h que indica si la altura del subárbol ha disminuido (h es verdadero si ha disminuido, falso en caso contrario). Sólo se investiga si hace falta reequilibrar cuando h es verdadero. Se asigna el valor verdadero a h cuando se encuentra y borra un nodo, o si al reequilibrar se reduce la altura de un subárbol.

Se introducen dos operaciones (simétricas) de equilibrado. Equilibra1 se emplea cuando la altura de la rama izquierda ha disminuido y Equilibra2 cuando la de la rama derecha ha disminuido. (Ver Niklaus Wirth. Algorithms+Structures=Programs.)

En un árbol equilibrado se pueden realizar en O(log n) unidades de tiempo, incluso en el peor de los casos, las siguientes operaciones:

- Encontrar un nodo con una llave dada
- Insertar un nodo con una llave dada
- Borrar un nodo con una llave dada

Estos resultados son consecuencias directas de un teorema demostrado por sus inventores.

METODO DE DISPERSION (HASH)

En los métodos de búsqueda dados anteriormente, se ve la necesidad de pasar a través de un número de llaves antes de hallar el registro que buscamos. En forma óptima nos gustaría tener una organización de archivo en la cual no se hacen comparaciones innecesarias. Precisamente de esto trata el presente método.

Este método consiste en tener una función H tal que H(k)=A, donde A es la dirección del registro cuya llave es k. Una función que transforma una llave en un índice de un archivo se llama función hash. De aquí que la aplicación de funciones hash utiliza la estructura de datos arreglo (ARRAY).

Cuando dos llaves, a través de la función H, se convierten al mismo entero surge lo que se denomina colisión.

Idealmente no debería ocurrir la situación anterior, pero en la práctica esto sucede comúnmente.

Tratarемos de desarrollar algunos métodos que se acercan al caso ideal y determinaremos qué acciones se deben tomar cuando no se logra el caso ideal.

FUNCIONES HASH

Describiremos algunos métodos para determinar funciones Hash:

Método de división

Una de las primeras funciones Hash es el método de división, el cual es definido como:

$$H(k)=k \bmod m + 1$$

para algún entero divisor m. Este método encuentra un "valor hash", el cual pertenece al conjunto (1,2,...,m).

Obviamente existen algunas m que dan mejores resultados, las cuales deben ser primas. Al emplear estas m se evitan situaciones en las que muchas llaves llegan a ocupar lugares ya utilizados (colisión).

Método del cuadrado medio

La llave es multiplicada por sí misma y los dígitos que aparecen en medio del producto son empleados como el índice. (El número de dígitos que se extrae para formar el índice depende de la cantidad permitida por éste.)

Ejemplo:

Sea k=123456, donde el cuadrado de la llave es 15241383936; si 3 dígitos son requeridos para la dirección, entonces las posiciones 5 hasta 7 son escogidas, lo cual da como resultado la dirección 138.

Método de dobles

Una llave es dividida en un número de partes, cada una de las cuales posee la misma longitud como requiere la dirección, con excepción posible de la última parte. Para obtener la dirección se suman las partes sin tomar en cuenta el acarreo final. Si las llaves tienen forma binaria, la operación de "OR exclusivo" puede ser sustituida por la operación suma.

Ejemplo:

Sea k=356942781 se transformará en una dirección de 3 dígitos.
Entonces 356 942 y 781 son sumados, encontrando la dirección 079.

SOLUCION A LAS COLISIONES

Cuando dos llaves distintas (k1 y k2) llevan a una misma localidad al aplicarles la función, se dice que hay una colisión. En seguida se dan algunos métodos para solucionar el problema en caso de colisiones, así como evaluaciones de cada técnica.

Una primera solución es tener un espacio en el archivo, por ejemplo al final, de modo que si ocurre una colisión el segundo registro se manda a esa sección; así al buscar este registro lo que se hace es ver si está en H(k), si no se busca por algún método en la sección de sobreflujo. Esta forma de hacerlo no es no es muy buena, ya que finalmente hay que efectuar varias comparaciones para encontrar lo que se buscaba.

DIRECCION ABIERTA

Si un registro con llave k es asignado a una dirección y ésta ya se halla ocupada, entonces el registro se coloca en la posición siguiente disponible en el arreglo.

Esta técnica se denomina prueba lineal; es un ejemplo del método general de resolver las colisiones, llamado reasignación, rehash (RH) o dirección abierta. En general la función de reasignación RH es aplicada al valor de H(k) para encontrar otra posición donde se pueda colocar el registro. Si la función RH(H(k)) se halla también ocupada, es transformada nuevamente con el fin de ver si RH(RH(H(k))) se encuentra disponible. Este proceso se continúa hasta que se localiza una posición vacía o la tabla ya está llena.

La desventaja que presenta este método es que asume una tabla fija de tamaño m. Si el número de registros aumenta más allá de m es imposible insertarlo sin que sea necesario asignar una tabla más grande y recalcular los valores de asignación de las llaves de todos los registros. Además es difícil eliminar un registro de esta tabla.

ENCADENAMIENTO

Consiste en mantener m listas ligadas una para cada posible dirección en la tabla Hash, siendo éstas los nodos de encabezamiento. Esto es que si hay una colisión en una dirección, todas las llaves que llevan a esta dirección se encadenan, de forma que al buscar un registro se accesa a la cabeza de la lista y se recorre la lista. Si no se encuentra el registro, se inserta al final de la lista.

Una ventaja de este método es la eliminación de nodos, pues se reduce simplemente a remover un nodo de la lista encadenada, esto es ajustar solamente unos pocos apuntadores. La desventaja es el espacio adicional que requieren los apuntadores.

El método de dispersión tiene mayor rendimiento con respecto a tiempo, incluso que las organizaciones más sofisticadas de árboles vistas, al menos en lo que se refiere a recuperación e inserción.

La mayor desventaja en relación con las técnicas de asignación dinámica de memoria es que el tamaño de la tabla es fijo, y no puede ajustarse según cambian las necesidades. Otra desventaja es en el borrado de elementos. Por lo tanto podemos decir que las organizaciones de árbol siguen siendo preferibles, cuando el volumen de los datos no se conoce bien o es muy variable.

CONCLUSIONES

Con los ejemplos dados podemos concluir que la representación y manipulación de las estructuras de datos resultan muy claras al utilizar el lenguaje PASCAL, debido a que los programas son fáciles de leer y entender y por ello constituyen una buena herramienta para la enseñanza de la programación.

A lo largo de este trabajo se ha hecho un esfuerzo por presentar de manera clara y concisa cada una de las estructuras de datos, a fin de formar un útil material de apoyo para la enseñanza y el aprendizaje.

Para un mayor aprovechamiento del tema, se recomiendan la implementación y el seguimiento detallado de los programas.

No obstante las ventajas del lenguaje PASCAL, éste tiene ciertas restricciones debido a que no ofrece mucha flexibilidad para manejar algunas estructuras de datos.

El trabajo está totalmente orientado al campo de la ciencia de la computación en un tema fundamental como es el procesamiento de datos, el cual constitutuye el eje principal de la computación.

BIBLIOGRAFIA

1.- PETER GROGONO
Programming in PASCAL
Addison-Wesley, 1980.

2.- AARON M. TENENBAUM y MOSHE J. AUGENTEIN
Data Structures Using PASCAL
Prentice-Hall, 1981.

3.- NIKLAUS WIRT
Algorithms + Data Structures = Programs
Prentice-Hall, 1976.

4.- JEAN-PAUL TREMBLAY y PAUL G. SORENSON
An Introduction to Data Structures with Applications
International Student, segunda edicion, 1984.

5.- ELLIS HOROWITZ y SARTAJ SAHNI
Fundamentals of Data Structures in PASCAL
Computer Science Press, 1984.

6.- G. H. GONNET
Handbook of Algorithms and Data (codificado en PASCAL y C)
Addison-Wesley, 1984.

7.- NELL DALE y SUSAN C. LILLY
PASCAL y Estructura de Datos
McGraw-Hill, 1986.

8.- THOMAS A. STANDISH
Data Structure Techniques
Addison-Wesley, 1980.

9.- JORGE I. EUAN A. y LUIS G. CORDERO B.
Estructura de Datos
UNAM - Facultad de Ingeneria, 1982.

10.- SEYMOUR LIPSCHUTZ
Estructura de Datos Serie SCHAUM
McGraw-Hill, 1987.

11.- KNUTH D. E.
The Art of Computer Programming, Vol. I y III
Addison-Wesley, 1973.

12.- NILS J. NILSSON
Principles of Artificial Intelligence
Tioga, 1980.