Estructuras de Datos e da Información

Kike Benlloch

1996

Índice

TDA 5

CONXUNTOS 5

COLAS 10

PILAS 13

RECURSIVIDADE 21

ARBORES 27

MONTICULOS 56

TDA

TDA (Tipo de Dato Abstracto) é unha abstracción matemática, un conxunto de operacións. Na definición do tipo de dato na sección type (morfoloxía do TDA) non se fai mención a como son esas operacións que definen o TDA, senón que estas van despois en forma de procedementos e funcións (sintaxe do TDA).
As avantaxes dun TDA son:

- -As operacións escríbense unha soa vez e son empregadas polos restantes módulos do programa.
- -As correcións nos módulos das operacións que definen o TDA son transparentes ó usuario e ós programas que empreguen o TDA.

CONXUNTOS

Un conxunto é un TDA (tipo de dato abstracto) que é unha colección de elementos na cal non hai elementos repetidos (un elemento aparece unha vez ou non aparece) e ademáis non están almacenados en ningúnha orde determinada.

exemplo:

{ 1 , 3 , 6 , 2 , 4 , 5 } é un conxunto
{ 1 , 2 , 3 , 4 , 5 , 6 } é o mesmo conxunto
{ 1 , 2 , 2 , 4 , 5 , 6 } non é un conxunto

Os conxuntos defínense mediante unha enumeración dos elementos:

$$X = \{ 1 , 2 , 3 , 4 \}$$

ou mediante unha serie de condicións que os definen:

$$X = \{ x \in Z / 1 <= x <= 4 \}$$

A relación básica é a de pertencia ($\in$) ou non dun elemento a un conxunto.
O conxunto baleiro é aquel que non tén elementos.
Dise que un conxunto A está incluído noutro conxunto B se tódolos elementos de A tamén pertencen a B.

operacións:

UNION	$A \cup B = \{ x / x \in A$ ou $x \in B \}$
INTERSECCION	$A \cap B = \{ x / x \in A$ e $x \in B \}$
DIFERENCIA	$A - B = \{ x / x \in A$ e $x \notin B \}$

Implementación dun conxunto mediante un vector booleano ("mapa de bits"):
Para representar un conxunto de enteiros 1..n pódese empregar un array de valores booleanos. Un elemento pertence ó conxunto se e so se o valor da súa compoñente é true. Esta representación pódese xeralizar para calquera tipo de conxunto e non so para conxuntos de enteiros.

```
type
     tipoelemento : 1 .. n ;
     conxunto : array [ tipoelemento ] of boolean ;
```

o conxunto { 1 , 4 } dentro do universo 1..6 represéntase como

```
|true|false|false|true|false|false|
   1    2     3     4    5     6      índices
```

Características:
-é un vector unidimensional onde os elementos gardan valores booleanos
-é válido cando hai un número finito de elementos
-os datos do conxunto deben ser datos simples de tipo ordinal

Avantaxes:
-operacións de unión e intersección realízanse de xeito moi rápido
-se o conxunto colle nunha palabra do ordenador (word) o ordenador realiza a operación dunha soa vez (procesa o conxunto dunha vez); as operacións polo tanto son rapidísimas

Inconvintes:
-limitación polo número finito de elementos
-deben estar nun subrango
-datos de tipo simple

Agora imos implementar algunha das funcións básicas para este tipo de conxuntos:

*Ca declaración TYPE que definimos arriba e a idea anterior, a inicialización a baleiro supón rechear tódalas compoñentes a false.

```
procedure conxuntobaleiro ( var a : conxunto ) ;
var
   i : integer ;
begin
     for i := 1 to n do { a complexidade da operación é n }
         a[i]:= false ;
end ;
```

*Para mirar se un conxunto é baleiro ou non buscamos a primeira compoñente que sexa true. Se non atopamos ningunha o conxunto é baleiro.

```
function e_baleiro ( a : conxunto ) : boolean ;
var
   i : integer ;
   nonbaleiro : boolean ;
begin
     i := 1 ;
     repeat { a complexidade da operación é n, aínda que en media é menos }
            nonbaleiro := a[i] ;
            i := i + 1 ;
     until (i>n) or nonbaleiro ;
     e_baleiro := not nonbaleiro ;
end ;
```

*A pertencia decídese simplemente mirando se a compoñente do elemento que buscamos está a true ou no. Análogamente insertar un elemento consiste en colocar a súa compoñente a true e borrar colocala a false.

```
function pertence ( a : conxunto ; i : tipoelemento ) : boolean ;
begin
     pertence := a[i] ; { a complexidade da operación é 1 }
end ;

procedure insertar ( var a : conxunto ; i : tipoelemento ) ;
begin
     a[i] := true ; { a complexidade da operación é 1 }
end ;

procedure borrar  ( var a : conxunto ; i : tipoelemento ) ;
begin
     a[i] := false ; { a complexidade da operación é 1 }
end ;
```

*Para a unión, intersección e diferencia utilizaremos os conectivos lóxicos

*Para a unión emprégase a disxunción (pertence a un ou pertence ó outro)

```
procedure union ( a , b : conxunto ; var c : conxunto ) ;
var
   i : integer ;
begin
     for i := 1 to n do
         c[i] := a[i] or b[i] ; { a complexidade da operación é n }
end ;
```

*Para a intersección emprégase a conxunción (pertence a un e pertence
ó outro)

```
procedure interseccion ( a , b : conxunto ; var c : conxunto ) ;
var
   i : integer ;
begin
     for i := 1 to n do
         c[i] := a[i] and b[i] ; { a complexidade da operación é n }
end ;
```

*Para a diferencia emprégase a conxunción e a negación (pertence a un e non
ó outro)

```
procedure diferencia ( a , b : conxunto ; var c : conxunto ) ;
var
   i : integer ;
begin
     for i := 1 to n do
         c[i] := a[i] and not b[i] ; { a complexidade da operación é n }
end ;
```

*Realización mediante un vector cos elementos.
Esta realización propón gardar tódolos elementos do conxunto nun vector
un detrás do outro, utilizando so as primeiras compoñentes necesarias.

```
const
     maxel = { estimación do máximo número de elementos posibles no conxunto }
type
    tipoelemento = { tipo de dato que se quere gardar no conxunto }
    conxunto = record
                     elementos : array [ 1 .. maxel ] of tipoelemento ;
                     cardinal : 0 .. maxel ;
               end ;
```

cardinal indica o derradeiro elemento do vector utilizado. Emprégase o nome
cardinal porque tamén indica o número de elementos que tén o conxunto.

```
exemplo:      1  2  3  ... maxel     índices
              1  4

              este é o conxunto {1,4}, cardinal val 2
```

*Desta maneira, o conxunto baleiro queda definido se colocamos cardinal a 0.
A decisión de se un conxunto é baleiro é agora inmediata.

```
procedure conxuntobaleiro ( var a : conxunto ) ;
begin
     a . cardinal := 0 ; { complexidade 1 }
end ;

function e_baleiro ( a : conxunto ) : boolean ;
begin
     e_baleiro := a . cardinal = 0 ; { complexidade 1 }
end ;
```

*Para determinar se un elemento pertence ou non ó conxunto, simplemente
faise unha búsqueda no vector dende a compoñente 1 ata a cardinal do
conxunto.

```
procedure insertar ( var a : conxunto ; i : tipoelemento ) ;
begin
     if not pertence ( a , i ) then
        with a do
             if cardinal = maxel
             then erro ( 'non hai sitio dispoñible' )
             else begin
                  cardinal := cardinal + 1 ; { complexidade n }
                  elemento [ cardinal := i ;
             end ;
end ;

function pertence ( a : conxunto ; i : tipoelemento ) : boolean ;
var
   j : integer ;
   esta : boolean ;
begin
     esta := false ;
     with a do
          while ( j <= cardinal ) and not esta do
                if elementos [ j ] = i
                then esta := true
                else j := j + 1 ;
     pertence := esta ;
end ;
```

*Cando se tenta insertar e non queda sitio dispoñible no vector para almacenalo prodúcese un erro. Este procedemento erro tamén se utilizará para a unión.

*Borrado: primeiro localízase o elemento dentro do vector. Conseguido isto reemprázase polo derradeiro elemento e decreméntase cardinal nunha unidade.

```
procedure borrar ( var a : conxunto ; i : tipoelemento ) ;
var
   j : integer ;
   esta : boolean ;
begin
     j := 0 ;
     esta := false ;
     with a do begin
          while ( j < cardinal ) and not esta do begin
                j := j + 1 ;                         { complexidade n }
                esta := elemento [ j ] = i ;
          end ;
          if esta then begin
             elementos [ j ] := elementos [ cardinal ] ;
             cardinal := cardinal - 1 ;
          end ;
     end ;
end ;
```

*Unión: todo o conxunto a e os elementos de b que non estean en a.

```
procedure union ( a , b : conxunto ; var c : conxunto ) ;
var
   i , j : integer ;
begin
     with c do begin
          for i := 1 to a . cardinal do
              elementos [ i ] := a . elementos [ i ] ;
          j := a . cardinal ;
          for i := 1 to b . cardinal do
          if not pertence ( a , b . elementos [ i ] ) then
             if j = maxel
             then erro ( 'non hai sitio dispoñible' )
             else begin
                  j := j + 1 ;   { complexidade card(a) + card(b) }
                  elementos [ j ] := b . elementos [ i ] ;
             end ;
          cardinal := j ;
     end ;
end ;
```

*Intersección: percórrese o conxunto a insertando en c so aqueles elementos que estean tamén en b.

```
procedure interseccion ( a , b : conxunto ; var c : conxunto ) ;
var
   i , j : integer ;
begin
     with c do begin
          j := 0 ;
          for i := 1 to a . cardinal do
              if pertence ( b , a . elementos [ i ] )
              then begin
                   j := j + 1 ;  { complexidade card(a) + card(b) }
                   elementos [ j ] := a . elementos [ i ] ;
              end ;
              cardinal := j ;
     end ;
end ;
```

*Diferencia : percórrese o conxunto a e gárdanse en c so os elementos que non estean en b.

```
procedure diferencia ( a , b : conxunto ; var c : conxunto ) ;
var
   i , j : integer ;
begin
     with c do begin
          j := 0 ;
          for i := 1 to a . cardinal do
              if not pertence ( b , a . elementos [ i ] )
              then begin
                   j := j + 1 ;  { complexidade card(a) + card(b) }
                   elementos [ j ] := a . elementos [ i ] ;
              end ;
              cardinal := j ;
     end ;
end ;
```

COLAS

As colas son un tipo de dato abstracto que se caracterízan por unha estructura FIFO. O primeiro elemento que entra é o primeiro que sae. A entrada á cola so se pode facer por un dos dous extremos e a saída polo outro extremo.

As colas caracterízanse polo xeito en que se tratan os elementos que hai nelas. Se accedemos a un elemento intermedio da cola inmediatamente deixa de ser unha cola porque non a estamos tratando como tal.

De cote chámase final da cola ó sitio polo que entran os elementos e principio ou cabeza por onde saen, como nunha cola normal.

A lectura nunha cola implica que o elemento lido sae da cola. Non se pode ler dúas veces o mesmo elemento.

Deque ou bicola é o TDA no que se pode producir a e/s polo principio ou polo final.
A bicola de entrada restrinxida é o TDA no que se pode sacar elementos polo principio ou polo final pero so se poden meter polo final.

As colas de prioridades soen ser colas de colas ordenadas por unha xerarquía arbitraria.
¡Nalgunhas implementacións de colas de prioridades dáse o fenómeno curioso de que se accede ós elementos intermedios! sen que isto supoña que se viola o concepto de cola.

IMPLEMENTACION ESTATICA DAS COLAS (en Pascal, mediante arrays)

PRIMEIRA IMPLEMENTACION
A entrada (final) é fixa mentres que a saída (principio) varía dentro do rango reservado.
-Avantaxes: sinxeleza.
-Inconvinte: limitación de memoria; desprazamento cada vez que se inserta un elemento na cola.

```
{----------------------------------------------------------------------------}
program
       colas ;
const
     ini = 1 ; { ¡chamo ini ó final da cola, por onde entran os elementos! }
     max = 5 ;
type
   tcola = record
                 arreglo : array [ ini .. max ] of char ;
                 cabeza : integer ;
           end ;
var
   cola : tcola ; c : char ; i : integer ;
{----------------------------------------------------------------------------}
procedure cls ;
          var
             i : integer ;
          begin
               for i := 1 to 24 do writeln ;
          end ;
{----------------------------------------------------------------------------}
procedure inicializar ( var cola : tcola ) ;
          begin
               cola . cabeza := ini - 1 ;
          end ;
```

```
{-------------------------------------------------------------------------}
function e_baleira ( cola : tcola ) : boolean ;
         begin
              e_baleira := cola . cabeza = ini - 1 ;
         end ;
{-------------------------------------------------------------------------}
function e_chea ( cola : tcola ) : boolean ;
         begin
              e_chea := cola . cabeza = max ;
         end ;
{-------------------------------------------------------------------------}
procedure insertar ( var cola : tcola ; c : char ) ;
          var
             i : integer ;
          begin
               if not e_chea ( cola ) then begin
                  for i := cola . cabeza downto ini do
                      cola . arreglo [ i + 1 ] := cola . arreglo [ i ] ;
                  cola . cabeza := cola . cabeza + 1 ;
                  cola . arreglo [ ini ] := c ;
               end ;
          end ;
{-------------------------------------------------------------------------}
procedure sacar ( var cola : tcola ) ;
          var
             i : integer ;
          begin
               if not e_baleira ( cola ) then begin
                  write ( 'Saiu o elemento: ' ) ;
                  writeln ( cola . arreglo [ cola . cabeza ] ) ;
                  cola . cabeza := cola . cabeza - 1 ;
               end ;
          end ;
{-------------------------------------------------------------------------}
procedure amosa_cola ( cola : tcola ) ;
          var
             i : integer ;
          begin
               for i := ini to cola . cabeza do
                   write ( cola . arreglo [ i ] , ' ' ) ;
               writeln ;
          end ;
{-------------------------------------------------------------------------}
begin
     inicializar ( cola ) ; randomize ; cls ;
     repeat
           write ( 'a cola é: ' ) ; amosa_cola ( cola ) ;
           writeln ( '1 - e_baleira' ) ;
           writeln ( '2 - e_chea' ) ;
           writeln ( '3 - insertar' ) ;
           writeln ( '4 - sacar' ) ;
           readln ( i ) ;
           case i of
                1 : writeln ( e_baleira ( cola ) ) ;
                2 : writeln ( e_chea ( cola ) ) ;
                3 : insertar ( cola , chr ( random ( 256 ) ) ) ;
                4 : sacar ( cola ) ;
           end ;
     until false ;
end .
```

SEGUNDA IMPLEMENTACION

Para evitar o desprazamento dos elementos da cola que se producía na anterior implementación cando se introducía un novo elemento, nesta segunda implementación tanto a fronte da cola (a cabeza, o inicio) como o final son índices variables.

A cola é circular: o seguinte ó derradeiro é o primeiro e o anterior ó primeiro é o derradeiro.

O problema que se plantexa é o de determinar cando a cola está baleira e cando está chea.

cola de 5 elementos, O é out (fronte) e I é in (final):

```
1 2 3 4 5   1 2 3 4 5   1 2 3 4 5   1 2 3 4 5   1 2 3 4 5   1 2 3 4 5
_ _ _ _ _   _ _ _ _ d   _ _ _ g d   _ _ v g d   _ k v g d   _ k v g _
      O I           I         I O       I   O     I     O     I   O
                    O
```

```
1 2 3 4 5   1 2 3 4 5  Vimos de cagala porque a situación inicial incitábanos
x k v g _   x k v g h  a pensar que cola_baleira := fronte = final - 1
I     O           O I  pero aquí está chea e esta condición cómprese.
```

Solucións posibles:

-Considerar que o rango válido é de 1 a n e a posición 0 é nula.
 Así, cando O + 1 = I (ousexa fronte é final -1) a cola está chea, e cando
 a cola está baleira tanto fronte como final apuntan a 0 (valen 0).

-Levar un simple contador de elementos que hai na cola.
-Perder unha celda, fronte apunta ó anterior do que hai que sacar e final
 apunta ó derradeiro que entrou.

 Inicializar : fronte := final := calquera posición válida do rango
 Baleira := final = fronte
 Chea := final + 1 = fronte

```
                                              (final=I fronte=O)
1 2 3 4 5   1 2 3 4 5   1 2 3 4 5   1 2 3 4 5   1 2 3 4 5   1 2 3 4 5
_ _ _ _ _   _ _ _ _ x   ñ _ _ _ x   ñ s _ _ x   ñ s h _ d   ñ s h _ _
      O           O I   I     O       I     O       I   O       I   O
      I                                          (cheo)
```

A implementación dinámica de colas faise mediante listas simplemente enlazadas nas que se inserta polo principio e se extrae polo final ou insértase polo final e se extrae polo principio.

Nas colas de prioridades existe un número asociado á información que indica un grao de xerarquía.

A implementación dunha cola de prioridades pode ser mediante unha lista ordenada por prioridades (¡dáse o fenómeno curioso de que se accede ós elementos intermedios!) ou mediante unha lista de colas. En calquera caso o primeiro criterio de ordenación é o de prioridade e o segundo o fifo.
A lista de colas é unha cola principal de nodos cabeceira contendo o número de prioridade, de cada un dos cales pendura unha cola fifo de elementos desa mesma prioridade.

```
Inserción nunha cola de prioridades:

SE a lista principal é baleira
ENT crear nodo cabeceira, gardar nel a prioridade do elemento que entra
    e crear un nodo único pendurando deste que garde ó elemento que entra
SENON
    SE a prioridade do que entra é maior que a do primeiro nodo cabeceira
    ENT insertar ó principio da lista de colas
    SENON
        MENTRES o seguinte cabeceira non sexa nulo
              E a prioridade do seguinte cabeceira sexa maior ou igual
                                                    que a do que entra
        FACER avanzar unha posición
                   ------------------------------------------------------
                   exemplo 10=maxprio 1=minprio insertar 3

                   ...-> 7 -> 5 -> X  ou ben ...-> 7 -> 3 -> X  ou ben
                              ^                         ^
                   ...-> 7 -> 5 -> 2 ->... ou ben ...-> 7 -> 5 -> 3 ->...
                              ^                                   ^
                   ------------------------------------------------------
        SE o actual cabeceira é da mesma prioridade que o que entra
        ENT insertar na cola que pendura de él
        SENON
            crear novo nodo cabeceira
            insertalo na lista principal entre o actual e o seguinte
            crear primeiro nodo que pendura do novo cabeceira e meter o novo
                                                                elemento
```

PILAS

As pilas son un tipo de dato abstracto que se caracterizan por unha estructura LIFO. O derradeiro elemento que entra é o primeiro que sae. A entrada e a saída so se poden facer por un dos dous extremos.

As pilas caracterízanse polo xeito en que se tratan os elementos que hai nelas. Se accedemos a un elemento intermedio da pila inmediatamente deixa de ser unha pila porque non a estamos tratando como tal.

Na memoria RAM hai unha parte que se chama stack que é unha pila onde se gardan as variables dos procesos. Para as variables globais resérvase espacio en tempo de compilación mentres que para as locais se reserva espacio en tempo de execución aínda que se definen en tempo de compilación. As variables locais so as permiten as linguaxes que teñen estructuras dinámicas.

O sitio por onde se inserta e se extrae chámase cabeza ou cumio da pila. Nalgunhas implementacións considérase unha operación de lectura non destructiva, ousexa mirar_cumio, que toma unha copia do elemento que está no cumio da pila sen que este saia da mesma. Pero normalmente a lectura dun elemento da pila implica que este elemento sae da mesma.

Nas implementacións dinámicas escollerase meter e sacar polo principio ou facer ambalas dúas cosas polo final. O normal é facelo polo principio xa que isto evita percorrer constantemente a pila ou manter un punteiro ó final da mesma.

```
{=============================================================================}
Exercicio:  Convertir unha pila de caracteres nun número real.
~~~~~~~~~~

       pila de caracteres
          1
          2
          3     => nºreal 123.5
          .
          5                  sen usar ningunha estructura auxiliar.


program
       pasa_pila_a_real ;

type
    tpila = ^ tnodopila ;

    tnodopila = record
                      car : char ;
                      sig : tpila ;
               end ;
{----------------------------------------------------------------------------}
procedure inipila ( var pila : tpila ) ;
begin
     pila := nil ;
end ;
{----------------------------------------------------------------------------}
function pilabaleira ( pila : tpila ) : boolean ;
begin
     pilabaleira := pila = nil ;
end ;
{----------------------------------------------------------------------------}
procedure metepila ( var pila : tpila ; c : char ) ;
var
   aux : tpila ;
begin
     if pilabaleira ( pila )
     then begin
          new ( pila ) ;
          pila ^ . car := c ;
          pila ^ . sig :=  nil ;
     end
     else begin
          new ( aux ) ;
          aux ^ . car := c ;
          aux ^ . sig := pila ;
          pila := aux ;
     end ;
end ;
{----------------------------------------------------------------------------}
procedure sacapila ( var pila : tpila ; var c : char ) ;
var
   aux : tpila ;
begin
     if not pilabaleira ( pila ) then begin
        c := pila ^ . car ;
        aux := pila ;
        pila := pila ^ . sig ;
        dispose ( aux ) ;
     end ;
end ;
```

```
{-----------------------------------------------------------------------}
procedure le_pila ( var pila : tpila ) ;
var
   str : string ;
   i : integer ;
begin
     writeln ( 'Escriba tódolos elementos da pila por orde dende o' ) ;
     writeln ( 'cumio ata o fondo, seguidos e sen espacios, e pulse Enter' ) ;
     readln ( str ) ;
     for i := length ( str ) downto 1 do
         metepila ( pila , str [ i ] ) ;
end ;
{-----------------------------------------------------------------------}
function convirte ( var pila : tpila ) : real ;
var
   c : char ;
   r : real ;
   parte_enteira : boolean ;
   divisor : real ;
begin
     r := 0 ;
     parte_enteira := true ;
     divisor := 10 ;
     while not pilabaleira ( pila ) do begin
           sacapila ( pila , c ) ;
           parte_enteira := parte_enteira and (c<>'.') ;
           if parte_enteira
           then r := r * 10 + (ord (c) - ord ('0'))
           else
               if c <> '.' then begin
                     r := r + ((ord (c) - ord ('0')) / divisor) ;
                     divisor := divisor * 10 ;
               end ;
     end ;
     convirte := r ;
end ;
{-----------------------------------------------------------------------}

var
   pila : tpila ;
begin
     le_pila ( pila ) ;
     writeln ( convirte ( pila ) ) ;
{
        exemplo da execución:
Escriba tódolos elementos da pila por orde dende o
cumio ata o fondo, seguidos e sen espacios, e pulse Enter
123.92837
 1.2392837000E+02
}
end.
```

```
{===========================================================================}
Exercicio: crear un conversor de expresións en notación infixa a notación
~~~~~~~~~~ prefixa e postfixa.

           -de prefixa a infixa, empezar polo final
           -de postfixa a infixa, empezar polo principio

           Ir ó final do código.

program
       convirte_de_infixa ;
const
     operadores = [ '*' , '+' , '-' , '/' ] ;
type
    tlista = ^ tnodolista ;

    tnodolista = record
                  car : char ;
                  sig : tlista ;
             end ;

    tpila = ^ tnodopila ;

    tnodopila = record
                  str : string ;
                  sig : tpila ;
              end ;

{---------------------------------------------------------------------------}
procedure metepila ( var pila : tpila ; str : string ) ;
var
   aux : tpila ;
begin
     if pila = nil
     then begin
          new ( pila ) ;
          pila ^ . str := str ;
          pila ^ . sig :=  nil ;
     end
     else begin
          new ( aux ) ;
          aux ^ . str := str ;
          aux ^ . sig := pila ;
          pila := aux ;
     end ;
end ;
{---------------------------------------------------------------------------}
procedure sacapila ( var pila : tpila ; var str : string ) ;
var
   aux : tpila ;
begin
     if pila <> nil then begin
        str := pila ^ . str ;
        aux := pila ;
        pila := pila ^ . sig ;
        dispose ( aux ) ;
     end ;
end ;
```

```
{--------------------------------------------------------------------------}
procedure metelista ( var lista : tlista ; c : char ) ;
var
   aux : tlista ;
begin
     if lista = nil then begin
        new ( lista ) ;
        lista ^ . car := c ;
        lista ^ . sig := nil ;
     end
     else begin
          new ( aux ) ;
          aux ^ . car := c ;
          aux ^ . sig := lista ;
          lista := aux ;
     end ;
end ;
{--------------------------------------------------------------------------}
procedure le_expr ( var lista : tlista ) ;
var
   str : string ;
   i : integer ;
begin
     writeln ( 'Escriba a expresión en infixa, e pulse Enter' ) ;
     readln ( str ) ;
     for i := length ( str ) downto 1 do
         metelista ( lista , str [ i ] ) ;
end ;
{--------------------------------------------------------------------------}
procedure le_expr_ó_reves ( var lista : tlista ) ;
var
   str : string ;
   i : integer ;
begin
     writeln ( 'Escriba a expresión, e pulse Enter' ) ;
     readln ( str ) ;
     for i := 1 to length ( str ) do
         metelista ( lista , str [ i ] ) ;
end ;
{--------------------------------------------------------------------------}
procedure in_a_pre ( in_ : tlista ; var pre : tpila ) ;
var
   str1 , str2 , str3 : string ;
begin
     while in_ <> nil do begin
           if ( in_ ^ . car <> '(' ) and ( in_ ^ . car <> ')' )
           then metepila ( pre , in_ ^ . car ) ;
           if in_ ^ . car = ')' then begin
              sacapila ( pre , str1 ) ;
              sacapila ( pre , str2 ) ;
              sacapila ( pre , str3 ) ;
              metepila ( pre , str2 + str3 + str1 ) ;
           end ;
           in_ := in_ ^ . sig ;
     end ;
     while pre ^ . sig <> nil do begin
              sacapila ( pre , str1 ) ;
              sacapila ( pre , str2 ) ;
              sacapila ( pre , str3 ) ;
              metepila ( pre , str2 + str3 + str1 ) ;
     end ;
end ;
```

```
{-------------------------------------------------------------------------}
procedure in_a_pos ( in_ : tlista ; var pos : tpila ) ;
var
   str1 , str2 , str3 : string ;
begin
     while in_ <> nil do begin
           if ( in_ ^ . car <> '(' ) and ( in_ ^ . car <> ')' )
           then metepila ( pos , in_ ^ . car ) ;
           if in_ ^ . car = ')' then begin
              sacapila ( pos , str1 ) ;
              sacapila ( pos , str2 ) ;
              sacapila ( pos , str3 ) ;
              metepila ( pos , str3 + str1 + str2 ) ;
           end ;
           in_ := in_ ^ . sig ;
     end ;
     while pos ^ . sig <> nil do begin
              sacapila ( pos , str1 ) ;
              sacapila ( pos , str2 ) ;
              sacapila ( pos , str3 ) ;
              metepila ( pos , str3 + str1 + str2 ) ;
     end ;
end ;
{-------------------------------------------------------------------------}

var
   lista : tlista ;
   pila : tpila ;
begin
     lista := nil ;
     le_expr ( lista ) ;
     in_a_pre ( lista , pila ) ;
     writeln ( 'En prefixa é: ' , pila ^ . str ) ;
     pila := nil ;
     in_a_pos ( lista , pila ) ;
     writeln ( 'En posfixa é: ' , pila ^ . str ) ;
end.

{ exemplo de execución deste programa:

Escriba a expresión en infixa, e pulse Enter
((a+b)*c)-(((f+h)+((l-g)*a))/(f-l))
En prefixa é: -*+abc/++fh*-lga-fl
En posfixa é: ab+c*fh+lg-a*+fl-/-

}

{==========================================================================}
POSFIXA A INFIXA
~~~~~~~~~~~~~~~~
                    A  B  C  *  D  E  F  ^  /  G  *  -  H  *  +

                    empézase polo principio, A

>-----------------------------------<
WHILE NOT COLABALEIRA DO

     [saca da cola]

     IF OPERANDO THEN METEPILA

     ELSE
          SACAPILA -> Y
          SACAPILA -> X
          METEPILA ( X OPERADOR Y )
>-----------------------------------<
```

exemplo:

```
A  B  C  *  D  E  F  ^  /  G  *  -  H  *  +
===========================================  é o estado inicial da cola
```

a evolución da pila é:

```
                   F
               E   E  E^F             G
    C      D   D   D   D   D/(E^F) D/(E^F)  (D/(E^F))*G
  B B B*C B*C B*C B*C B*C    B*C     B*C       B*C     B*C - (D/(E^F))*G
A A A  A   A   A   A   A      A       A         A              A
- - - --- --- --- --- ---  ------- ------- ----------- -----------------

       H
B*C - (D/(E^F))*G  ( B*C - (D/(E^F))*G )* H
       A                      A              A + ( ( B*C - (D/(E^F))*G )* H )
-----------------  ------------------------  --------------------------------
                                             a expresión final hai que
                                             sacala da pila tralo bucle
```

```
{=============================================================================}
PREFIXA A INFIXA
~~~~~~~~~~~~~~~~
                    A  B  C  *  D  E  F  ^  /  G  *  -  H  *  +

                    empézase polo final, +

>-----------------------------------<
WHILE NOT COLABALEIRA DO

     [saca da cola]

     IF OPERANDO THEN METEPILA

     ELSE
          SACAPILA -> X
          SACAPILA -> Y
          METEPILA ( X OPERADOR Y )
>-----------------------------------<

{=============================================================================}
INFIXA A POSFIXA
~~~~~~~~~~~~~~~~
                    (1.2+(4/2))*(2.3-(21.1*6.3))

                    empézase polo principio, 1.2

>-----------------------------------<
WHILE NOT COLABALEIRA DO

     [saca da cola]

     IF )
         SACAPILA -> Y
         SACAPILA -> OPERADOR
         SACAPILA -> X
         SACAPILA (
         METEPILA X,Y,OPERADOR

     ELSE METEPILA
>-----------------------------------<
```

exemplo:

```
(1.2+(4/2))*(2.3-(21.1*6.3))
============================ é o estado inicial da cola
```

a evolución da pila é:

```
                       2
                   /   /
               4   4   4
           (   (   (   (    4,2,/
       +   +   +   +   +      +                                    (
  1.2 1.2 1.2 1.2 1.2 1.2    1.2                     *             *
(  (   (   (   (   (   (      (     1.2,4,2,/,+  1.2,4,2,/,+  1.2,4,2,/,+
- --- --- --- --- --- --- ) ----- ) -----------  -----------  -----------

                                                                     6.3
                                                        *            *
                                         21.1         21.1          21.1
                              (            (            (            (
                 -            -            -            -            -
   2.3          2.3          2.3          2.3          2.3          2.3
    (            (            (            (            (            (
    *            *            *            *            *            *
1.2,4,2,/,+  1.2,4,2,/,+  1.2,4,2,/,+  1.2,4,2,/,+  1.2,4,2,/,+  1.2,4,2,/,+
-----------  -----------  -----------  -----------  -----------  ----------- )


21.1,6.3,*
    -
   2.3
    (         2.3,21.1,6.3,*,-
    *               *
1.2,4,2,/,+     1.2,4,2,/,+
----------- ) ----------------  ou ben se obriga a que toda a expresión
                                estea rodeada por derradeiros parénteses, ou
                                ben hai que repetir unha vez a operación de
                                sacar da pila, tralo bucle. O resultado é:
                                       1.2,4,2,/,+,2.3,21.1,6.3,*,-,*

{============================================================================}
INFIXA A PREFIXA
~~~~~~~~~~~~~~~~
                    (1.2+(4/2))*(2.3-(21.1*6.3))

                    empézase polo principio, 1.2

>----------------------------------<
WHILE NOT COLABALEIRA DO

     [saca da cola]

     IF )
         SACAPILA -> Y
         SACAPILA -> OPERADOR
         SACAPILA -> X
         SACAPILA (
         METEPILA OPERADOR,X,Y

     ELSE METEPILA
>----------------------------------<
```

RECURSIVIDADE

A recursividade aplícase cando para solucionar un problema, este se pode dividir noutros máis pequenos e todos se resolven do mesmo xeito.

Hai recursividade cando un subprograma se chama a sí mesmo directa ou indirectamente:
-recursividade directa: no código que conforma dito subprograma existe unha chamada a sí mesmo.
-recursividade indirecta: existe unha secuencia de chamadas entre subprogramas a cal ó final se convirte nun lazo pechado.

¿Cando hai que aplicar a recursividade?
-cando as estructuras de datos usadas se definen de maneira recursiva.
-cando un problema se autodefine de maneira recursiva.
-cando un problema se divide en subprogramas máis pequenos que precisan o mesmo tratamento.

TODO problema resolto de forma recursiva se pode resolver por un método iterativo.

As implementacións recursivas soen ser máis claras pero menos eficientes cás iterativas.

A recursividade so é posible en linguaxes que permitan asignación dinámica de memoria, pois nun programa recursivo non se sabe de antemán cantas chamadas se van facer ó algoritmo (p.ex., no cálculo do factorial non se sabe cantas variables se van utilizar):

A's variables locais resérvaselles espacio de memoria en tempo de execución.

Hai que ter tino de que o programa recursivo non se chame a sí mesmo indefinidamente e tén que haber un camiño ou condición co cal non se volve chamar ó caso base. E' dicir:

As normas son

1ª) Inclusión do caso base: no código debe aparecer o caso "parvo" do problema, o que tén resolución inmediata. No exemplo do factorial, if n = 0 then fact := 1.
2ª) Reducción do problema: debemos comprobar que en cada paso o problema se achega á solución, que queda menos traballo por facer. No exemplo do factorial, fact := n * factorial (n-1).
3ª) Rexeitamento de parámetros non válidos: antes de aplicar a recursividade débese confirmar que os parámetros son axeitados. No exemplo do factorial, if n >= 0 then begin ...

O que estamos consumindo na aplicación da recursividade é espacio da pila de memoria (STACK). Aínda que o "nome" da variable é o mesmo, estamos gardando valores distintos, variables diferentes (distintas direccións de memoria) referidas a distintos niveis de chamada.

O' usar un método iterativo e non o recursivo, emprégase o HEAP en troques do stack.

RECURSION DE COLA é aquela que se produce cando un subprograma se chama a sí mesmo na derradeira liña do código.

```
{═══════════════════════════════════════════════════════════════════════════}
Exercicio: Dado un número enteiro, crear un procedemento recursivo que
~~~~~~~~~~ escriba os díxitos dese número en orde.

{---------------------------------------------------------------------------}

program numero ; uses crt ;

procedure escribe ( n : longint ) ;
begin
     if n div 10 = 0
     then write ( n )
     else begin
          escribe ( n div 10 ) ; write ( '  ' , n mod 10 ) ;
     end ;
end ;

begin clrscr ;
     escribe ( 110293 ) ;
end.

{ na pantalla aparece: 1  1  0  2  9  3 }

{---------------------------------------------------------------------------}
```

Vemos que esta solución é pouco eficiente porque se van deixando operacións pendientes. A solución iterativa sería máis rápida. A conversión dun programa recursivo a iterativo faise siempre mediante bucles (e.g. while) e o uso de pilas.

So se pode pasar de recursividade a iteración sen usar pilas cando a chamada recursiva vai ó final do código do subprograma recursivo.

```
{═══════════════════════════════════════════════════════════════════════════}
```

Outro exemplo do pouco eficiente que pode chegar a ser un programa recursivo: os números de Fibonacci.

```
function fibo ( n : integer ) : integer ;
begin
     if n > 0
     then
       if (n = 1) or (n = 0)
       then fibo := n
       else fibo := fibo ( n-1 ) + fibo ( n-2 )
     else fibo := 0 ;
end ;

exemplo:                                   fibo(5)
                            fibo(4)           +             fibo(3)
                    fibo(3)    +   fibo(2)          fibo(2)  +    fibo(1)
               fibo(2)+fibo(1)  fibo(1)+fibo(0)  fibo(1)+fibo(0)
           fibo(1)+fibo(0)
```

Xa para un número pequeno como 5 se acumulan moitos cálculos repetidos.

```
{═══════════════════════════════════════════════════════════════════════════}
Exercicio: Construir un recoñecedor de palíndromos.
~~~~~~~~~~

function palindromo ( s : string ) : boolean ;
begin
     if length ( s ) < 2
     then palindromo := true
     else palindromo := ( s [ 1 ] = s [ length (s) ] )
                        and
                        ( palindromo ( copy ( s , 2 , length (s) -2 ) ) ) ;
end ;
```

```
{════════════════════════════════════════════════════════════════════════════════════════}
Exercicio: procedemento copiador de lista simplemente enlazada.
~~~~~~~~~~
procedure copia ( lista1 : tlista ; var lista2 : tlista ) ;
begin
     if lista1 = nil
     then lista2 := nil
     else begin
         new ( lista2 ) ;
         lista2 ^ . info := lista1 ^ . info ;
         copia ( lista1 ^ . sig , lista2 ^ . sig ) ;
     end ;
end ;
```

Supoñendo o mesmo procedemento pero:

```
procedure copia ( lista1 , lista2 : tlista ) ;
```

(sen var) e referido a listas con nodo cabeceira ¿funcionaría?
A resposta é que sí.

```
{════════════════════════════════════════════════════════════════════════════════════════}
Exercicio: Función copiadora de lista simplemente enlazada.
~~~~~~~~~~
function copia ( lista : tlista ) : tlista ;
var
   c : tlista ;
begin
     if lista = nil
     then c := nil
     else begin
         new ( c ) ;
         c ^ . info := lista ^ . info ;
         c ^ . sig := copia ( lista ^ . sig ) ;
     end ;
     copia := c ;
end ;
```

A L G O R I T M O S   D E   B A C K T R A C K I N G (VOLTA ATRAS):

{══}
Exercicio: as oito raíñas: atopar como deben disporse nun taboleiro para
~~~~~~~~~~ que non se maten mutuamente.

```
X _ _ _ _ _ _ _
_ _ X _ _ _ _ _
_ _ _ _ X _ _ _
_ _ _ _ _ _ X _
_ X _ _ _ _ _ _
_ _ _ X _ _ _ _
_ _ _ _ _ X _ _
_ _ _ _ _ _ _ X
```

{está en seudocódigo}

```
procedure coloca ( i : integer { fila } ; var q : boolean ) ;
{ para cada raíña busca unha posición correcta en cada fila }
var
   j : integer ;
begin
     j := 0 ; { columna }
     q := false ;
     repeat
           j := j + 1 ;
           if e_valido ( i , j ) then
                coloca ( i , j ) ;
                if i < 8 then
                     coloca ( i+1 , q ) ;
                     if not q then quitar ( i , j )
                else q := true ;
     until (j = 8) or q ;
end ;
```

{══}
Exercicios propostos:
~~~~~~~~~~~~~~~~~~~~~
1) salto do cabalo: se a casiña inicial dun cabalo de xadrez é aleatoria ¿pode visitar tódalas casiñas do taboleiro unha e so unha vez?
2) laberinto construido cunha matriz 20x20 de booleans, 0 paso ceibe, 1 paso cortado, situación inicial aleatoria ¿pódese sair?
3) función comparadora de dúas listas simplemente enlazadas.

=-=

ORDEACION POR MISTURA

Divídese o array en dúas partes. Ordénanse por separado ambalas dúas partes e mistúranse. A subdivisión do array repítese ata que cada subarray tén un so elemento, i.e. está ordeado.

{está en seudocódigo}

```
procedure ord_mistura ( var datos : arreglo ; primeiro , derradeiro : integer
) ;
var
   metade : integer ;
begin
     if primeiro < derradeiro then
     begin
          metade := ( primeiro + derradeiro ) div 2 ;
          ord_mistura ( datos , primeiro , metade ) ;
          ord_mistura ( datos , metade +1 , derradeiro ) ;
          mistura ( datos , primeiro , metade , metade +1 , derradeiro ) ;
          { percorre as dúas partes tomando o menor de cada par e
          despois emborcando o contido }
     end ;
end ;
```

Este algoritmo supón unha mellora en canto á velocidade: tén complexidade
$n \log 2 (n)$ onde $n < n \log 2 (n) < n^2$
O único inconvinte que tén é que usa un array auxiliar para almacenar datos.

ORDEACION RAPIDA

```
procedure division ( var datos : arreglo ; primeiro , derradeiro : integer ;
var puntodivision : integer ) ;
{
Separa os elementos do arreglo datos entre os índices primeiro e derradeiro
segundo sexan os elementos menores ou maiores que puntodivisión.
Tómanse dous índices, un ó principio e outro ó final.
}
var
   dereita , esquerda , v : integer ;
begin
     v := datos [ primeiro ] ;
     dereita := primeiro + 1 ;
     esquerda := derradeiro ;

     repeat
           { mover dereita ata elemento > v }
           while ( dereita < esquerda ) and ( datos [ dereita ] <= v )
           do dereita := dereita + 1 ;
           { comprobar condición de fin }
           if ( dereita = esquerda ) and ( datos [ dereita ] <= v )
           then dereita := dereita + 1 ;
           { mover esquerda ata elemento <= v }
           while ( dereita <= esquerda ) and ( datos [ esquerda ] > v )
           do esquerda := esquerda - 1 ;
           if dereita < esquerda then begin
              trocar ( datos [ esquerda ] , datos [ dereita ] ) ;
              dereita := dereita + 1 ;
              esquerda := esquerda - 1 ;
           end ;
     until dereita > esquerda ;

     trocar ( datos [ primeiro ] , datos [ esquerda ] ) ;
     puntodivisión := esquerda ;
end ;
```

No mellor dos casos prodúcense n-1 comparacións, daquela no mellor
dos casos a complexidade do algoritmo de división é n.

```
procedure ord_rapida ( x , i , j ) ;
var
   p : integer ;
begin
     if i < j then begin
        division ( x , i , j  , p ) ;
        ord_rapida ( x , i , p-1 ) ;
        ord_rapida ( x , p+1 , j ) ;
     end ;
end ;
```

```
chamadas arrays
   0       1
   1       2
   2       4
   3       8
      ...
```

A complexidade de ord_rapida é lg 2 (n)

Entón a complexidade do algoritmo de ordenación rápida é de n¨lg 2 (n)
no mellor dos casos (i.e. os subarrays teñen o mesmo tamaño)

=-=

```
function busca_nodo ( d : tdato ; l : tlista ) : tnodo ;
var
   atopado : boolean ;
begin
     atopado := false ;
     while l <> nil and not atopado do
           if ( l ^ . info . info1 = d . info . info1 )
           and ( l ^ . info . info2 = d . info . info2 )
           then atopado := true
           else l := l ^ . sig ;
           busca_nodo := l ;
end ;

procedure ins_nodo ( d : tdato ; var l : tlista ) ;
var
   ptr : tlista ;
   atopado : boolean ;
begin
     if listabaleira ( l ) then begin
        crearnodo ( d , ptr ) ;
        l := ptr ;
     end
     else begin
          ptr := busca_nodo ( d , l ) ;
          if ptr <> nil
          then ptr ^. rep := ptr ^ . rep + 1 ;
          else begin
               atopado := false ;
               if ( l ^ . info . info1 > d . info . info1 )
               and ( l ^ . info . info2 > d . info . info2 )
               then
               ...
          end ;
     end ;
end ;
```

ARBORES

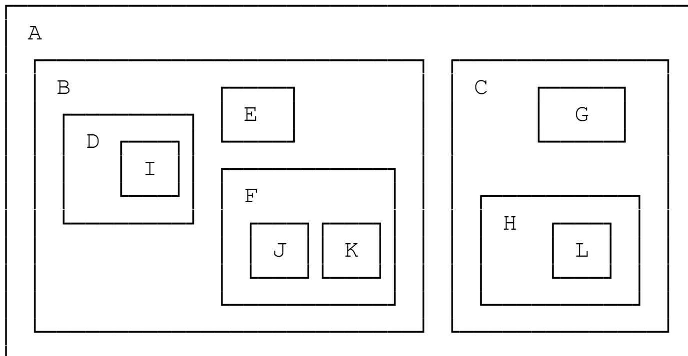
Def.- Unha árbore tipo T é unha estructura homoxénea que é a concatenación dun elemento de tipo T cun número finito de árbores disxuntas chamadas subárbores. Unha forma particular de árbore pode ser a estructura baleira. Disxunto significa que so hai unha forma posible de chegar a cada punto da árbore.

Unha árbore é unha lista non lineal.
-Lista porque contén elementos homoxéneos.
-Non lineal porque dado un elemento, este pode ter cero, un ou máis dun sucesor (elemento "seguinte"). O antecesor é único.

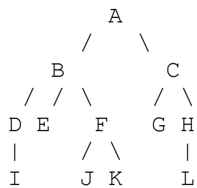
Podemos definir unha árbore como un TDA formado por un elemento diferenciado chamado raiz e por varias árbores que suceden ó elemento raiz (definición recursiva de árbore)

representacións:

a) diagrama de Venn



b) grafo



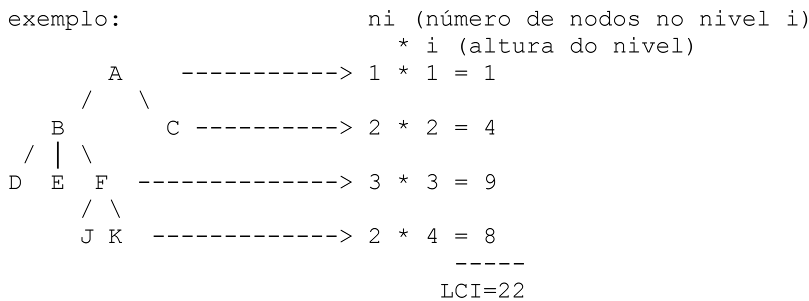
Características (segundo Cairó)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-toda árbore non baleira tén un único NODO RAIZ.
-un nodo X é DESCENDENTE DIRECTO ou FILLO dun nodo Y se o nodo X é apuntado polo nodo Y.
-un nodo X é ANTECESOR DIRECTO ou PAI dun nodo Y se o nodo X apunta ó nodo Y.
-dise que tódolos nodos que son descendentes directos dun mesmo nodo (fillos dun mesmo pai) son IRMANS.
-todo nodo que non tén ramificacións chámase NODO TERMINAL ou FOLLA.
-todo nodo non raiz nin terminal dise INTERIOR.
-GRAO DUN NODO é o número de descendentes directos do mesmo.
-GRAO DUNHA ARBORE é o número máximo de descendentes directos dun nodo calquera da árbore.
-NIVEL é o número de arcos que deben ser percorridos para chegar a un determinado nodo. Por definición a raiz tén nivel 1.
-ALTURA da árbore é o máximo número de niveis de tódolos nodos da árbore. (o nivel máximo de tódolos nodos da árbore)
-BOSQUE é un conxunto de ó menos unha árbore disxunta.

Defínese lonxitude de camiño como o número de arcos que deben ser percorridos para chegar dende a raiz a determinado nodo.

LONXITUDE DE CAMIÑO INTERNO
~~~~~~~~~~~~~~~~~~~~~~~~~~~
L.C.I.: E' a suma das lonxitudes de camiño de tódolos nodos da árbore. Pode calcularse por medio da seguinte fórmula:

$$L.C.I. = \sum_{i=1}^{h} n_i * i$$

h : altura da árbore
n_i : número de nodos no nivel i
i : nivel da árbore (altura do nivel)



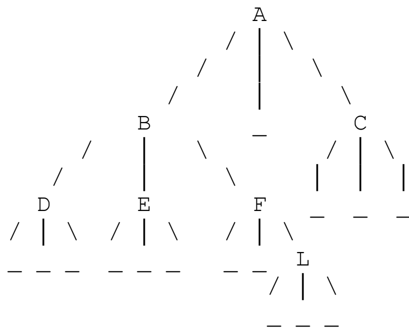
Agora ben, a Media de Lonxitude de Camiño Interno L.C.I.M. calcúlase dividindo a L.C.I. entre o número de nodos existentes na árbore:

$$L.C.I.M. = \frac{L.C.I.M.}{n}$$

no exemplo LCIM sería 22/8

LONXITUDE DE CAMIÑO EXTERNO
~~~~~~~~~~~~~~~~~~~~~~~~~~~

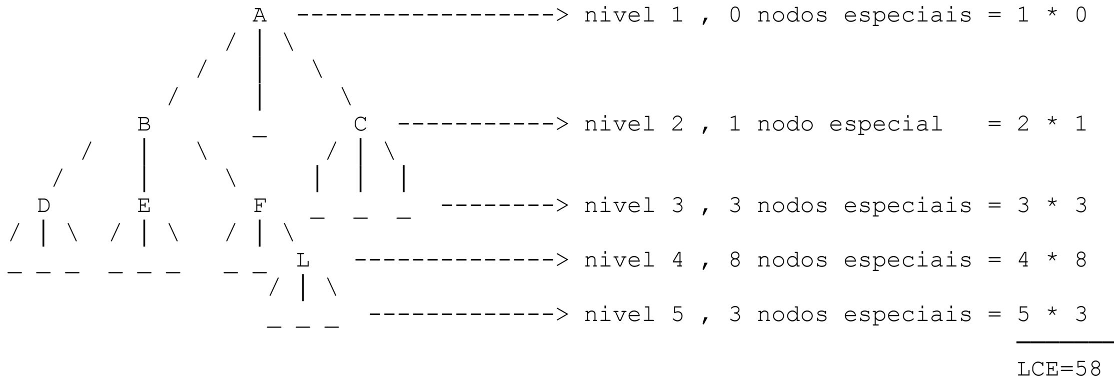
Arbore extendida é aquela na que o número de fillos de cada nodo é igual ó grao da árbore. Se algún dos nodos da árbore non compre esta condición debe incorporárselle ó mesmo nodos especiais, tantos como sexan necesarios.



L.C.E.: E' a suma das lonxitudes de camiño de tódolos nodos ESPECIAIS da árbore. Calcúlase por medio da seguinte fórmula:

$$L.C.E. = \sum_{i=2}^{h+1} ne_i * i$$

exemplo:



A Media de Lonxitude de Camiño Externo L.C.E.M. calcúlase así:

$$L.C.E.M. = \frac{L.C.E.M.}{ne}$$

no exemplo sería 58/15

ARBORES BINARIAS
~~~~~~~~~~~~~~~~

Unha árbore ordenada é aquela na que as ramas dos nodos da árbore están ordenadas.
Unha árbore binaria é unha árbore ordenada de grao 2.

Def.- Unha árbore binaria de tipo T é unha estructura homoxénea que se compón mediante a concatenación dun elemento de tipo T chamado raiz, con dúas árbores binarias disxuntas chamadas subárbore esquerda e subárbore dereita da raiz.

As árbores binarias non son directamente comparables cas árbores ordinarias.

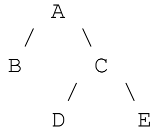
ARBORES BINARIAS DISTINTAS son aquelas con estructuras diferentes
~~~~~~~~~~~~~~~~~~~~~~~~~~ (independientemente da información que se
garda nos nodos).

ARBORES BINARIAS SEMELLANTES son aquelas con estructuras idénticas
~~~~~~~~~~~~~~~~~~~~~~~~~~~~ pero a información que conteñen é distinta.

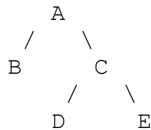
ARBORES BINARIAS EQUIVALENTES son aquelas que teñen igual estructura e
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ contido.

exemplo:

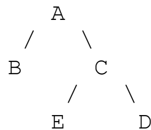
dada a árbore:



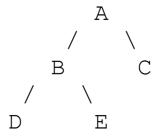
é EQUIVALENTE:



é SEMELLANTE:



é DISTINTA:



Arbores binarias completas: tódolos seus nodos, agás os do derradeiro nivel, teñen dous fillos.

Número máximo de nodos dunha árbore con altura h é= 2^{h} -1

Número máximo de nodos no nivel i dunha árbore é= 2¨i -1

Lema: se n0 é o número de nodos terminais e n2 o número de nodos de grao 2, daquela n0 = n2 + 1. Demostralo.

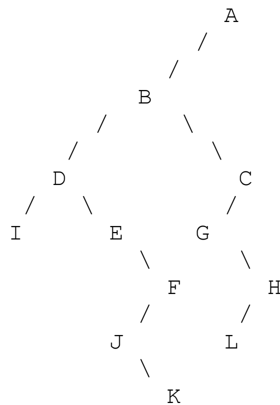
-se a árbore é baleira n0 = 0 e n2 = 0 daquela non se compre

-se a árbore non é binaria non se compre

Entón as hipóteses son que a árbore é non baleira e binaria.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
--demostración:          sexa n1 o número de nodos de grao 1
                         sexa B o número de conexións existentes na árbore

número de nodos da árbore n = n0 + n1 + n2 (a)

obviamente n = B + 1 (para unir n nodos son necesarias n-1 aristas)

como por cada nodo de grao 1 hai unha conexión e por cada nodo de grao 2 hai
dúas conexións daquela: B = n1 + 2 * n2

destas dúas expresións dedúcese que: n - 1 = n1 + 2 * n2 //
// n = n1 + 2 * n2 + 1 (b)

de (a) - (b) dedúcese:

n = n0 + n1 + n2 (a)
n = n1 + 2 * n2 + 1 (b)
________________________
0 = n0 + n2 - 2 * n2 -1 // 0 = n0 - n2 -1 // n0 = n2 + 1    c.q.d.
                                             ~~~~~~~~~~~
para toda árbore binaria non baleira

XERALES COMO BINARIAS
~~~~~~~~~~~~~~~~~~~~~
               conversión de árbore multicamiño (grao>2) en binaria (grao=2)

pasos a seguir:

a) débese enlazar os fillos de cada nodo en forma horizontal.
b) débese enlazar en forma vertical o nodo pai co fillo que se
atopa máis á esquerda. Ademáis, debe eliminarse o vínculo dese
pai co resto dos seus fillos.
c) débese virar toda a estructura.

exemplo:
--a árbore orixinal é:

                 A
               /   \
             /       \
           /           \
         B               C
     /   |   \         /   \
   /     |     \      |     |
  D      E       F    G     H
  |            /   \        |
  I           J     K       L

--tralos pasos 1 e 2 queda:

                 A
               /
             /
           /
         B ------------> C
     /                 /
   /                  |
  D ---> E ----> F    G --> H
  |            /            |
  I           J --> K       L
```

--tralo paso 3 queda:



REPRESENTACION DE ARBORES BINARIAS EN MEMORIA
~~~~~~
Existen dúas formas tradicionais de representar unha árbore binaria en memoria:
--mediante arreglos
--por medio de datos de tipo punteiro

Neste derradeiro caso empréganse os seguintes recursos:

```
type
     tipoinfo = ...
     tipoarbore = ^ tiponodo ;
     tiponodo = record
                      info : tipoinfo ;
                      fillo_esq , fillo_der : tipoarbore ;
                end ;
var
   raiz : tipoarbore ;
```

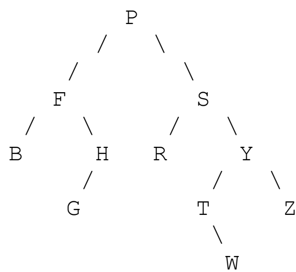
PERCORRIDOS EN ARBORES BINARIAS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Consiste en visitar os nodos da árbore dun xeito sistemático de modo que os nodos sexan visitados todos e cada un unha soa vez.

1.-Percorrido en INORDE:
     -percorrer a subárbore esquerda
     -visitar a raiz
     -percorrer a subárbore dereita

2.-Percorrido en PREORDE:
     -visitar a raiz
     -percorrer a subárbore esquerda
     -percorrer a subárbore dereita

3.-Percorrido en POSTORDE:
     -percorrer a subárbore esquerda
     -percorrer a subárbore dereita
     -visitar a raiz

exemplo:

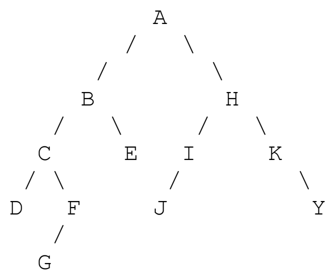



```
Inorde: B F G H P R S T W Y Z

Preorde: P F B H G S R Y T W Z

Postorde: B G H F R W T Z Y S P

Inorde invertida: Z Y W T S R P H G F B
( D , raiz , E )

Postorde invertida: Z W T Y R S G H B F P
( D , E , raiz )
```

```
PERCORRIDOS RECURSIVOS
{-----------------------------------}
procedure inorde ( a : tipoarbore ) ;
begin
     if ptr <> nil then begin
          inorde ( a ^ . fillo_esq ) ;
          procesar ( a ^ . info ) ;
          inorde ( a ^ . fillo_der ) ;
     end ;
end ;
{------------------------------------}
procedure preorde ( a : tipoarbore ) ;
begin
     if ptr <> nil then begin
          procesar ( a ^ . info ) ;
          preorde ( a ^ . fillo_esq ) ;
          preorde ( a ^ . fillo_der ) ;
     end ;
end ;
{-------------------------------------}
procedure postorde ( a : tipoarbore ) ;
begin
     if ptr <> nil then begin
          postorde ( a ^ . fillo_esq ) ;
          postorde ( a ^ . fillo_der ) ;
          procesar ( a ^ . info ) ;
     end ;
end ;
{-------------------------------------}
```

```
PERCORRIDOS ITERATIVOS
{--------------------------------------------}
procedure inorde ( raizarbore : tipoarbore ) ;
var
   pilaptr : tipopila ;
   ptr : tipopunteiro ;
begin
     limparpila ( pilaptr ) ;
     ptr := raizarbore ;
     repeat
          { procesar ata que se acabe a árbore }
          { tamén podería ser while (ptr<>nil) }
          { and not pilabaleira(pila) then ... }
          { ir á esquerda todo canto se poida }
          while ptr <> nil do begin
               meterpila ( pilaptr , ptr ) ;
               ptr := ptr ^ . fillo_esq ;
          end ; { while }
          { se existe algo na pila, sacar, procesar e mover á dereita }
          if not pilabaleira ( pila ) then begin
               sacar ( pilaptr , ptr ) ;
               procesarnodo ( ptr ) ;
               ptr := ptr ^ . fillo_der ;
          end ; { if }
     until ( ptr = nil ) and pilabaleira ( pilaptr ) ;
end ; { procedure }
{---------------------------------------------}
procedure preorde ( raizarbore : tipoarbore ) ;
var
   pilaptr : tipopila ;
   ptr : tipopunteiro ;
begin
     limparpila ( pilaptr ) ;
     ptr := raizarbore ;
     repeat
          { procesar ata que se acabe a árbore }
          { tamén podería ser while (ptr<>nil) }
          { and not pilabaleira(pila) then ... }
          { ir á esquerda todo canto se poida }
          while ptr <> nil do begin
               procesarnodo ( ptr ) ;
               meterpila ( pilaptr , ptr ) ;
               ptr := ptr ^ . fillo_esq ;
          end ; { while }
          { se existe algo na pila, sacar e mover á dereita }
          if not pilabaleira ( pila ) then begin
               sacar ( pilaptr , ptr ) ;
               ptr := ptr ^ . fillo_der ;
          end ; { if }
     until ( ptr = nil ) and pilabaleira ( pilaptr ) ;
end ; { procedure }
{--------------------------------------------}
```

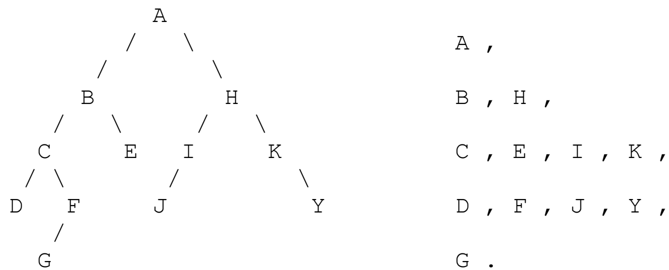
Para o procedemento iterativo para a postorde son necesarias dúas pilas. Procésase o nodo gardado tras acceder ó lado dereito. Recupérase e non se procesa senón que se vai á dereita e a segunda vez que se recupera procésase. As partes esquerdas son directamente procesables; as dereitas, non.




```
PILA a     PILA b

 -D          S
  F          N
 -C          S
  E          N
 -B          S
  H          N
 -A          S
```

```
se raiz <> nil
     repetir
          mentres raiz <> nil
               meterpila ( raiz , true )
               se raiz ^ . fillo_der <> nil
                    meterpila ( raiz ^ . fillo_der , false )
               finse
               raiz := raiz ^ . fillo_esq
          finmentres
          bPila := false

          mentres non bPila
               se pilabaleira ( raiz )
                    bPila := true
               senon
                    se sacarpila ( raiz )
                         procesar ( raiz )
                    senon
                         bPila := true
                    finse
               finse
          finmentres
     ata pilabaleira ( raiz )
```

PERCORRIDO POR ANCHURA: Os nodos da árbore percórrense por nivel. Emprégase unha cola.




```
procedure anchura ( raiz : tipoarbore ) ;
begin
     limparcola ( cola ) ;
     if raiz <> nil
     then insertarcola ( raiz , cola ) ;
     while not colabaleira ( cola ) do begin
          sacarcola ( raiz ) ;
          procesarcola ( raiz ) ;
          if raiz ^ . fillo_esq <> nil
          then insertarcola ( raiz ^ . fillo_esq , cola ) ;
          if raiz ^ . fillo_der <> nil
          then insertarcola ( raiz ^ . fillo_der , cola ) ;
     end ; { while }
```

```
{---------------------------------------------------------------------------}
Exercicio: crear un procedemento que permita crear unha árbore en memoria a
~~~~~~~~~~ partir dunha árbore baleira.
A secuencia a seguir será a seguinte:
-primeiro pregúntase a información que se gardará no nodo.
-a continuacion pregúntase se vai a existir outro nodo con parte esquerda e
se é así introdúcese o seu valor.
-sobre ese novo nodo repítese a consulta. En caso de non ter parte
esquerda pregúntase se tén parte dereita.

Deseñalo recursivamente.

Supóñense dous casos:
-nodo raiz (parámetro por valor) xa existe (non contén ningunha información).
-nodo raiz non existe (parámetro por referencia porque hai que apuntar ó
nodo raiz que se cree)

{-------------------------------------}
procedure crear ( raiz : tipoarbore ) ;
var
   op : char ;
begin
     writeln ( 'Dato? ' ) ;
     readln ( raiz ^ . info ) ;
     writeln ( 'Tén fillo esquerdo? ' ) ;
     readln ( op ) ;
     if op = 's' then begin
          new ( raiz ^ . esq ) ;
          crear ( raiz ^ . esq ) ;
     end
     else raiz ^ . esq := nil ;
     { end if-then-else }
     writeln ( 'Tén fillo dereito? ' ) ;
     readln ( op ) ;
     if op = 's' then begin
          new ( raiz ^ . der ) ;
          crear ( raiz ^ . der ) ;
     end
     else raiz ^ . der := nil ;
     { end if-then-else }
end ; { procedure }
{-----------------------------------------}
procedure crear ( var raiz : tipoarbore ) ;
var
   op : char ;
begin
     new ( raiz ) ;
     writeln ( 'Dato? ' ) ;
     readln ( raiz ^ . info ) ;
     writeln ( 'Tén fillo esquerdo? ' ) ;
     readln ( op ) ;
     if op = 's'
     then crear ( raiz ^ . esq ) ;
     else raiz ^ . esq := nil ;
     writeln ( 'Tén fillo dereito? ' ) ;
     readln ( op ) ;
     if op = 's'
     then crear ( raiz ^ . der ) ;
     else raiz ^ . der := nil ;
end ; { procedure }
```

```
{---------------------------------------------------------------------------}
Exercicio: crear unha función que pasándolle unha árbore binaria e un certo
~~~~~~~~~~ dato tipo info indique se ese dato está almacenado nalgún nodo da
árbore. A' función pásaselle a árbore, o dato e devolve un valor booleano.

function esta_en_arbore ( a : arbore ; d : tipoinfo ) : boolean ;
begin
     if a = nil
     then esta_en_arbore := false
     else esta_en_arbore := ( a ^ . info = d )
                        or esta_en_arbore ( a ^ . esq , d )
                        or esta_en_arbore ( a ^ . der , d ) ;
end ; { function }

{---------------------------------------------------------------------------}
Exercicio: función recursiva que permite facer un duplicado dunha árbore.
~~~~~~~~~~ Pásaselle a árbore e devolve o punteiro á raiz da copia.

function copiar ( raiz : tipoarbore ) : tipoarbore ;
var
   novo : tipoarbore ;
begin
     if raiz = nil
     then copiar := nil
     else begin
          new ( novo ) ;
          novo ^ . info := raiz ^ . info ;
          novo ^ . esq := copiar ( raiz ^ . esq ) ;
          novo ^ . der := copiar ( raiz ^ . der ) ;
          copiar := novo ;
     end ; { else }
end ; { function }

{---------------------------------------------------------------------------}
Exercicio: función recursiva que permite facer un duplicado dunha árbore
~~~~~~~~~~ pero de modo que sexa como a imaxe reflexada nun espello.
           Pásaselle a árbore e devolve o punteiro á raiz da árbore
           "reflexado" (problema da imaxe especular).

function espello ( raiz : tipoarbore ) : tipoarbore ;
var
   novo : tipoarbore ;
begin
     if raiz = nil
     then copiar := nil
     else begin
          new ( novo ) ;
          novo ^ . info := raiz ^ . info ;
          novo ^ . esq := espello ( raiz ^ . dta ) ;
          novo ^ . dta := espello ( raiz ^ . esq ) ;
          espello := novo ;
     end ; { else }
end ; { function }

NOTA: advertir que é practicamente igual ó anterior.

{---------------------------------------------------------------------------}
Exercicio: función recursiva que devolve o número de nodos dunha árbore.
~~~~~~~~~~
function contanodos ( arbore : tarbore ) : integer ;
begin
     if arbore <> nil
     then contanodos := 1 + contanodos ( arbore ^ . esq )
                           + contanodos ( arbore ^ . dta )
     else contanodos := 0 ;
end ;
```

```
{---------------------------------------------------------------------------}
Exercicio: función recursiva que devolve a altura dunha árbore de grao 3.
~~~~~~~~~~
function altura3 ( raiz : tipoarbore ) : integer ;
begin
     if raiz = nil
     then altura := 0
     else
         altura := 1 + maximo ( altura ( raiz ^ . esq ) ,
                                altura ( raiz ^ . ctr ) ,
                                altura ( raiz ^ . dta ) ) ;
end ; { function }
```

(queda por facer a función máximo de tres números)

ARBORES ESPECIAIS
~~~~~~~~~~~~~~~~~
Arbores de expresións aritméticas: nodos con fillos de 0 ou 2 fillos.
Os nodos con (2) fillos corresponden a operadores e os nodos sen fillos
a operandos.

exemplo:

```
    *
   / \
  -   C
 / \
A   B
```

A representación é directa e en infixa non require parénteses.

-función que devolva a avaliación dunha expresión nunha árbore
-¿qué dous percorridos necesito para establecer a estructura dunha árbore binaria?
 prefixa e infixa, ver (*)
-función que pasándolle unha árbore binaria devolve o nivel da árbore que máis nodos contén e o número de nodos dese nivel.

ARBORES ESPECIFICAS
~~~~~~~~~~~~~~~~~~~
Arbores binarias de búsqueda: Estructura non lineal na cal cada elemento tén 0, 1 ou 2 sucesores establecéndose unha relación de orde entre un sucesor, o elemento e o outro sucesor.

Def.- Para todo nodo dun certo tipo dunha árbore de búsqueda:
-débese establecer entre os nodos da árbore algún tipo de relación de orde.
-en base á relación tódolos nodos á esquerda dun dado deben ser menores ou iguais ca este e os da dereita maiores ou iguais ca este.

A estructura dunha árbore binaria de búsqueda depende da implementación e da secuencia de entrada dos datos.

(*) exemplo:

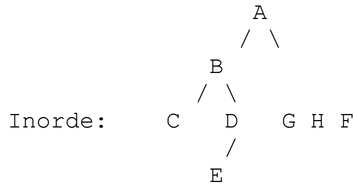
se temos dous percorridos seguintes dunha árbore binaria de búsqueda:
Inorde: C B E D A G H F
Preorde: A B C D E F G H   (nota: a orde alfabética non intervén)

A preorde indica que a raiz da árbore é A, daquela na inorde
"tiramos" de A cara arriba:

```
                 A
                / \
Inorde: C B E D   G H F
```

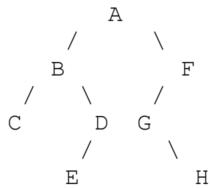
Repítese o proceso recursivamente cos subárbores esquerdo e dereito

A preorde indica que a seguinte raiz é B, repetimos o proceso:



```
                A
               / \
             B
            / \
Inorde:    C   D   G H F
               /
             E
```

Quédanos por resolver a subárbore dereita de A:
Inorde: G H F
Preorde: F G H
Tiramos de F que é a raiz e quédanos (segundo indica a inorde) a subárbore esquerda por resolver):
Inorde: G H (esquerda - raiz - dereita)
Preorde: G H (raiz - esquerda - dereita)   H queda á dereita de G
O resultado é:



```
       A
    /     \
   B        F
 /   \    /
C     D  G
    /     \
   E        H
```

E' necesario ter a preorde.

Exercicio: atopar un nodo nunha árbore binaria de búsqueda.

```
{-----------------------------------------------------------------}
procedure buscar {iterativamente}
( raiz : tipoarbore ; valor : tipoinfo ; var atopado : boolean ) ;

var
   ptr : tipoarbore ;
begin
     ptr := raiz ;
     atopado := false ;
     while ( ptr <> nil ) and ( not atopado ) do
          if ( ptr ^ . info = valor )
          then atopado := true
          else
               if ( ptr ^ . info > valor )
               then ptr := ptr ^ . esq
               else ptr := ptr ^ . dta ;
end ; { procedure }
{-----------------------------------------------------------------}
procedure buscar {recursivamente}
( raiz : tipoarbore ; valor : tipoinfo ; var atopado : boolean ) ;

var
   ptr : tipoarbore ;
begin
     if raiz = nil
     then atopado := false
     else
          if ( raiz ^ . info = valor )
          then atopado := true
          else
               if ( raiz ^ . info > valor )
               then buscar ( raiz ^ . esq , valor , atopado )
               else buscar ( raiz ^ . dta , valor , atopado ) ;
end ; { procedure }
{-----------------------------------------------------------------}
```

```
Inserción de nodos en árbores binarias de búsqueda
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1) comparar a clave a insertar ca raiz da árbore. Se é maior débese
avanzar cara a subárbore dereita. Se é menor débese mover á subárbore
esquerda.
2) repetir sucesivamente o paso anterior ata que se cumpra algunha das
seguintes condicións:
        -A subárbore dereita é igual á baleira; ou a subárbore esquerda é
         igual ó baleira. Insértase no lugar axeitado.
        -A clave que se quere insertar é igual á raiz da árbore.
         Non se realiza inserción.

{----------------------------------------------------------------}
procedure insertar ( var raiz : tipoarbore ; valor : tipoinfo ) ;
{ implementación recursiva }
begin
     if ( raiz = nil ) then begin
        new ( raiz ) ;
        raiz ^ . info := valor ;
        raiz ^ . esq := nil ;
        raiz ^ . dta := nil ;
     end
     else
        if ( raiz ^ . info > valor )
        then insertar ( raiz ^ . esq , valor )
        else insertar ( raiz ^ . dta , valor )
        { recursión de cola }
end ; { procedure }
{---------------------------------------------------------------}
{ implementación iterativa }
var
   novo , ptr , anterior : tipopunteiro ;
   clavenova : tipoclave ;
begin
     new ( novo ) ;
     novo ^ . esq := nil ;
     novo ^ . dta := nil ;
     novo ^ . info := info ;
     clavenova := info ;

     ptr := raiz ;
     anterior := nil ;
     while ptr <> nil do begin
        anterior := ptr ;
        if ptr ^ . info > clavenova
        then ptr := ptr ^ . esq
        else ptr := ptr ^ . dta
     end ; { while }

        if anterior = nil
        then raiz := novonodo
        else
            if anterior ^ . info . clave > clavenova
            then anterior ^ . esq := novonodo
            else anterior ^ . dta := novonodo
end ; { insertar iterativo }
```

Supresión de nodos en árbores binarias de búsqueda

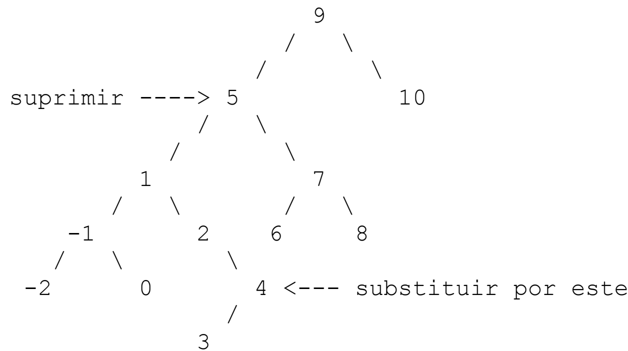
primeiro hai que encontrar o nodo que queremos eliminar. Despois consiste en eliminar un nodo da árbore binaria de búsqueda sen violar os principios que precisamente o definen. Hai os seguintes casos, en función de se o número de fillos é 0, 1 ou 2:

-se o elemento a borrar é terminal ou folla, simplemente suprímese.

-se o elemento a borrar tén un so descendente, sutitúese por ese descendente.

-se o elemento a borrar tén dous descendentes, sutitúese polo nodo que se atopa máis á dereita da subárbore esquerda.

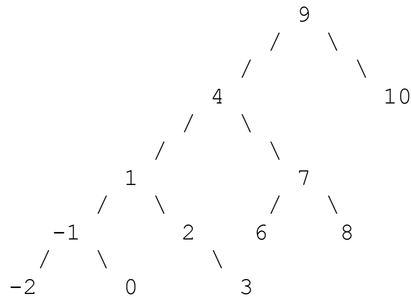
exemplo:



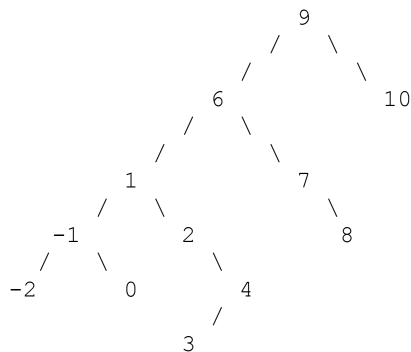
Do que se trata é de localizar o elemento maior da subárbore esquerda do que vamos a borrar. Tamén poderíamos localizar o elemento menor da subárbore dereita.

E' dicir facemos o movemento 1 á esquerda, todo á dereita (ou na outra opción faríamos 1 á dereita, todo á esquerda) e logo "pontéase" do pai ó fillo do elemento que vimos de borrar.

Ca primeira solución quedaría:



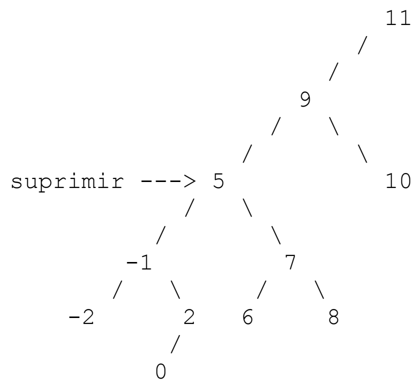
Ca segunda solución quedaría:

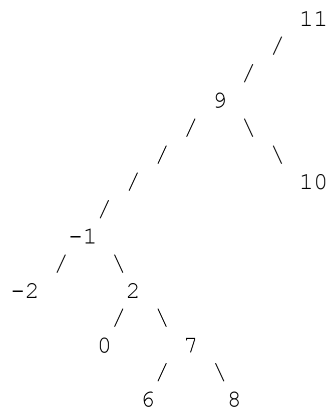


```
procedure suprimir ( var raizarbore : tipoarbore ; valorclave : tipoclave ) ;
var
   ptr , anterior : tipopunteiro ;
begin
     ptr := raizarbore ; anterior := nil ;
     while ptr ^ . info . clave > valorclave do begin
          anterior := ptr ;
          if ptr ^ . info . clave > valorclave
          then ptr := ptr ^ . esquerdo
          else ptr := ptr ^ . dereito ;
     end ; { while }
     { suprimir nodo apuntado por ptr ; anterior apunta ó pai dese nodo }
     suprimirnodo ( raizarbore , ptr , anterior ) ;
end ;

procedure suprimirnodo ( var raiz : tipopunteiro ; ptr , anterior :
tipopunteiro ) ;
{
Suprime o nodo apuntado por ptr; anterior é un punteiro para o nodo pai, ou
é nil se o nodo a suprimir é a raiz
}
var
   temp : tipopunteiro ;
begin
     { caso de supresión dunha folla }
     if ( ptr ^ . dereito = nil ) and ( ptr ^ . esquerdo = nil )
     then
          if anterior = nil
          then raiz := nil
          else
              if anterior ^ . dereito = ptr
              then anterior ^ . dereito := nil
              else anterior ^ . esquerdo := nil
     else
          { caso de supresión de nodo con dous fillos }
          if ( ptr ^ . dereito <> nil ) and ( ptr ^ . esquerdo <> nil ) then
begin
             {
             Buscar o valor a reemprazar. Atopar o nodo que contén o
             valor máis próximo ó que se vai a suprimir
             }
             anterior := ptr ;
             temp := anterior ^ . esquerdo ;
             while temp ^ . dereito <> nil do begin
                  anterior := temp ;
                  temp := temp ^ . dereito ;
             end ;
             { copiar a información }
             ptr ^ . info := temp ^ . info ;
             if anterior = ptr
             then anterior ^ . esquerdo := temp ^ . esquerdo
             else anterior ^ . dereito := temp ^ . esquerdo ;
             .
             .
             .
```

```
             if ptr ^ . dereito <> nil
             then
                 if anterior = nil
                 then raizarbore := ptr ^ . dereito
                 else
                     if anterior ^ . dereito = ptr
                     then anterior ^ . dereito := ptr ^ . dereito
                     else anterior ^ . esquerdo := ptr ^ . dereito
             else
                 if anterior = nil
                 then raizarbore := ptr ^ . esquerdo
                 else
                     if anterior ^ . dereito = ptr
                     then anterior ^ . dereito := ptr ^ . esquerdo
                     else anterior ^ . esquerdo := ptr ^ . esquerdo ;
             dispose ( ptr )
          end ; { if }
end ; { suprimirnodo }
```

Unha solución alternativa para a supresión dun nodo nunha árbore binaria de búsqueda:





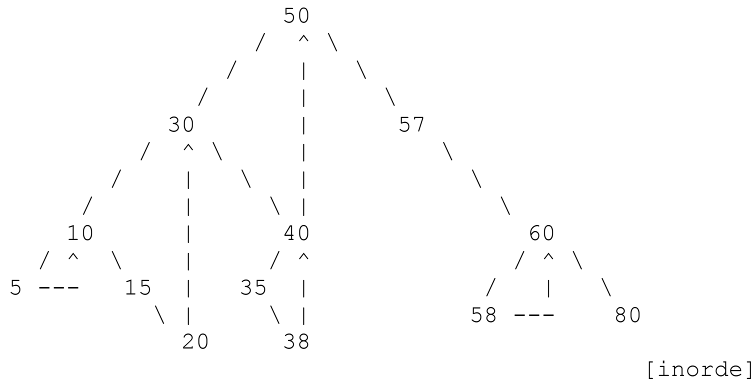
{ implementación iterativa }

```
{ implementación recursiva }
procedure eliminar ( var raiz : tiponodo ; valor : info ) ;
var
   aux : tiponodo ;

          procedure delete ( var r : tiponodo ) ;
          begin
               if ( r ^ . dereita <> nil )
               then delete ( r ^ . dereita )
               else begin
                    aux ^ . info := r ^ . info ;
                    aux := r ;
                    r := r ^ . esq ;
               end ; { else }
          end ; { delete }

begin { eliminar }
     if raiz <> nil
     then
         if ( valor < raiz ^ . info )
         then eliminar ( raiz ^ . esquerda , valor )
         else
             if ( valor > raiz ^ . info )
             then eliminar ( raiz ^ . dereita , valor )
             else begin
                 aux := raiz ;
                 if ( raiz ^ . dereita = nil )
                 then raiz := aux ^ . esquerda
                 else
                     if ( raiz ^ . esquerda = nil )
                     then raiz := aux ^ . dereita
                     else delete ( aux ^ . esquerda ) ;
             end ; { else }
             dispose ( aux ) ;
end ; { eliminar }
```

OUTRAS ESTRUCTURAS DE TIPO ARBORE
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Arbores entrelazadas: melloran os percorridos.
Arbores balanceadas: melloran o rendemento en operacións de búsqueda.
Tries: empréganse para a implementación de diccionarios.
Arbores multicamiño: empréganse para tomas de decisión.
Montículos: empréganse para operacións de ordenación.
Complexidade n¨lg 2 (n), melloran a ordenación por mistura.
Empréganse ás veces para implementar colas de prioridade.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ARBORES BINARIAS ENTRELAZADAS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Melloran as operacións de percorrido sobre todo a de percorrido en profundidade. A idea consiste en eliminar a pila.

Os nodos compóñense de cinco campos:
-info
-2 punteiros: dereita, esquerda
-2 valores booleanos. Se o campo booleano é true, o punteiro dereita ou esquerda segundo sexa o caso apunta a un antecesor e non a un descendente, neste caso apunta ó seu antecesor seguinte no percorrido inorde.



esta é unha árbore entrelazada pola dereita: ó non ser necesarias as chamadas recursivas afórrase tempo e memoria.

en xeral o tipo nodo é:

L th	Esq	Info	Der	R th

Implementación:

```
type
     tipo_nodo = ^ nodo ;
     nodo = record
                 info : tipoinfo ;
                 esq , der : tiponodo ;
                 rth : boolean ;
            end ;
```

segundo esta implementación (coincide ca do exemplo) o nodo é así:

Esq	Info	Der	R th

```
{─────────────────────────────────────────────────────────────────────────────}
function arbore ( x : tipoinfo ) : tiponodo ;
var
   ptr : tiponodo ;
begin
     new ( ptr ) ;
     ptr ^ . info := x ;
     ptr ^ . esq := nil ;
     ptr ^ . der := nil ;
     ptr ^ . rth := false ;
end ;
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
{————————————————————————————————————————————————————————————————————————————}
{ percorrido en inorde para árbores entrelazadas pola dereita }
procedure inorde ( a : entrelazada_der ) ;
begin
     if a <> nil then begin
       while a ^ . dereita <> nil do begin
          while a ^ . esq <> nil do
               a := a ^ . esq ;
          procesa ( a ) ;
          a := a ^ . der ;
       end ;
       procesa ( a ) ;
     end ;
end ;
{————————————————————————————————————————————————————————————————————————————}
procedure insertaresquerda ( p : tiponodo ; x : tipoinfo ) ;
var
   q : tiponodo ;
begin
     if ( p <> nil ) then begin
       if ( p ^ . esq <> nil ) then begin
          q := arbore ( x ) ;
          p ^ . esquerda := q ;
          q ^ . dereita := p ;
          q ^ . rth := true ;
       end ;
     end ;
end ;
{————————————————————————————————————————————————————————————————————————————}
procedure insertardereita ( p : tiponodo ; x : integer ) ;
var
   q , r : tiponodo ;
begin
     if p <> nil then begin
       if ( p ^ . dereita <> nil ) then
         if p ^ . rth { non tén fillos á dereita } then begin
           q := arbore ( x ) ;
           r := p ^ . dereita ;
           p ^ . rth := false ;
           p ^ . dereita := q ;
           q ^ . dereita := r ;
           q ^ . rth := true ;
         end ; { non tén fillos á dereita }
         if ( p ^ . dereita = nil ) then begin
           q := arbore ( x ) ;
           p ^ . dereita := q ;
         end ; { derradeiro nodo }
     end ; { if p <> nil }
end ; { insertadereita }

{ procesamento de árbore enlazada en inorde }
{ evítanse a recursividade e as pilas }
```

```
procedure inorde ( tree : tiponodo ) ;
var
   p , q : tiponodo ;
begin
     p := tree ;
     repeat
           q := nil ;
           while ( p <> nil ) do begin
                q := p ;
                p := p ^ . esquerda ;
           end ; { while }
           if ( q <> nil ) then begin
             write ( q ^ . info ) ;
             p := q ^ . dereita ;
             while ( q ^ . rth ) do begin
                  write ( p ^ . info ) ;
                  q := p ;
                  p := p ^ . dereita ;
             end ; { while }
           end ; { if }
     until ( q = nil ) ;
end ; { inorde }
```

ARBORES BALANCEADAS (AVL)
~~~~~~~~~~~~~~~~~~~~~~~~~~~

Nas árbores binarias de búsqueda pódense realizar eficientemente as operacións de búsqueda, inserción e eliminación de nodos. Sen embargo, se a árbore medra ou decrece descontroladamente o rendemento pode disminuir considerablemente.

Con obxecto de mellorar o rendemento na búsqueda xorden as árbores balanceadas (AVL). A idea é a de realizar reacomodos ou balanceos despois de insercións ou eliminacións de elementos.

Def.- Defínese unha árbore balanceada como unha árbore binaria de búsqueda na cal se debe de cumprir a seguinte condición:

"para cada nodo T da árbore, a altura das subárbores esquerdo e dereito non debe diferir en máis dunha unidade"

A complexidade deste tipo de árbore en canto ó tempo é moi superior ás anteriores árbores.

As árbores balanceadas teñen unha certa similitude, en canto ó seu mecanismo de formación, ós números de Fibonacci. A árbore de altura 0 é baleira, a árbore de altura 1 tén un único elemento e en xeral o número de nodos dunha árbore con altura h > 1 calcúlase mediante a seguinte fórmula:

$$k_h = k_{h-1} + 1 + k_{h-2}$$

onde k é o número de nodos e h a altura da árbore.

Para poder determinar se unha árbore está balanceada ou non, debe manexarse información relativa ó equilibrio de cada nodo da árbore (factor de equilibrio dun nodo, FE).

O FE defínese como a altura da subárbore dereita menos a altura da subárbore esquerda.

$$FE = H_{rd} - H_{re}$$

onde r* é a rama * e H_{r*} é a altura da rama *

Os valores que pode tomar FE son -1, 0, ou 1 e -2 ou 2 na situación previa a reestructurar a árbore.
~~~~~~~~~~~~~~~~~~~~~~~~~~~

Inserción en AVL

~~~~~~~~~~~~~~~~

1.- Localizar o lugar onde hai que insertar o elemento.
2.- Calcular o seu FE, que será 0 e regresar polo camiño de búsqueda calculando o FE dos distintos nodos.
3.- Se algún de dichos nodos viola o criterio de equilibrio, débese reestructurar a árbore.
4.- O proceso remata ó chegar á raiz da árbore ou cando se realiza a reestructuración do mesmo, neste caso non é necesario determinar o FE dos restantes nodos.

Caso 1.- As ramas esquerda re e dereita rd teñen a mesma altura ...
- 1.1.- ... e insértase en re : Hre > Hrd (árbore balanceada)
- 1.2.- ... e insértase en rd : Hrd > Hre (árbore balanceada)

Caso 2.- As ramas esquerda re e dereita rd teñen distinta altura ...
- 2.1.- ... e Hre < Hrd ...
  - 2.1.1.- ... e insértase en re : Hrd = Hre (árbore balanceada)
  - 2.1.2.- ... e insértase en rd : REEQUILIBRAR
- 2.2.- ... e Hre > Hrd ...
  - 2.2.1.- ... e insértase en re : REEQUILIBRAR
  - 2.2.2.- ... e insértase en rd : Hrd = Hre (árbore balanceada)

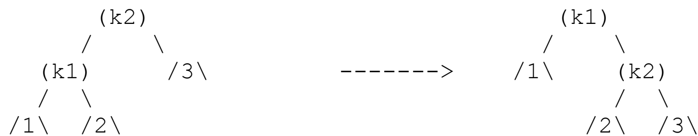
REEQUILIBRAR consiste en rotar os nodos da árbore.

Rotación
- Simple (2 nodos)
  - DD fillo dereito da raiz pasa a ser a raiz
  - EE fillo esquerdo da raiz pasa a ser a raiz
- Composta (3 nodos)
  - DE
  - ED

===

Rotación simple: dous nodos cambian de posición:

Rotación EE :

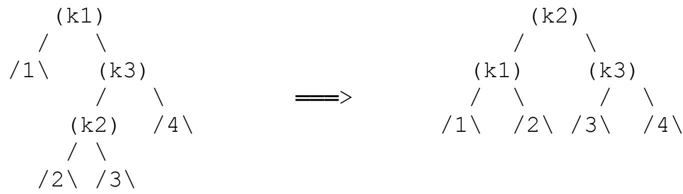


/i\ son subárbores dos que a forma non importa

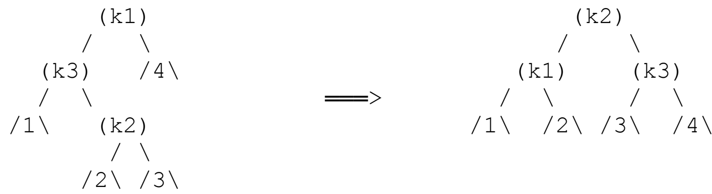
Rotación DD :

/i\ son subárbores dos que a forma non importa
~~~~~~~~~~~~~~~~

==
Rotación composta: tres nodos cambian de posición:
----
a rotación DE (dereita->esquerda) é en xeral así:

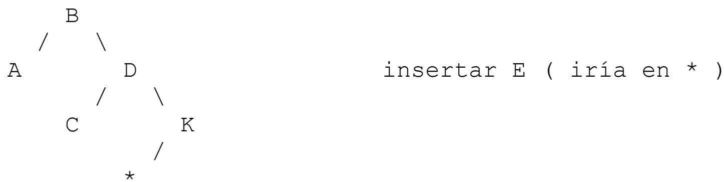


/i\ son subárbores dos que a forma non importa
----
a rotación ED (esquerda->dereita) é en xeral así:

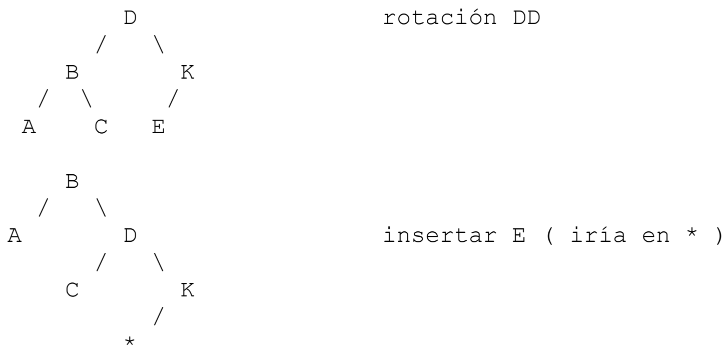


/i\ son subárbores dos que a forma non importa
----

outro exemplo:

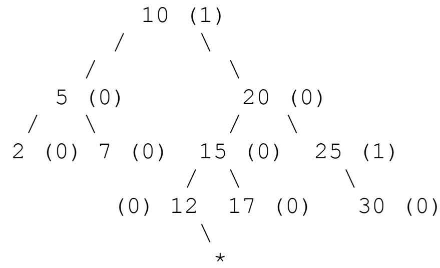


o resultado é:

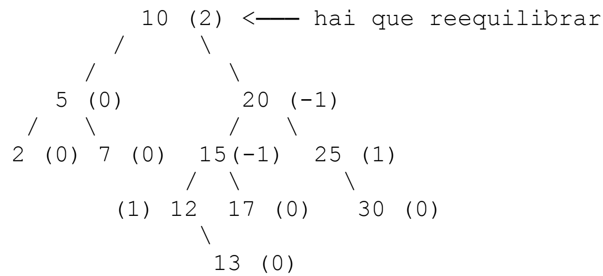


Insertar o 13 no seguinte árbore AVL: Iría en *

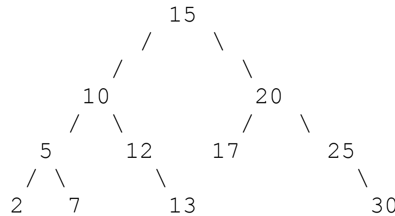
entre paréntеses vai o FE



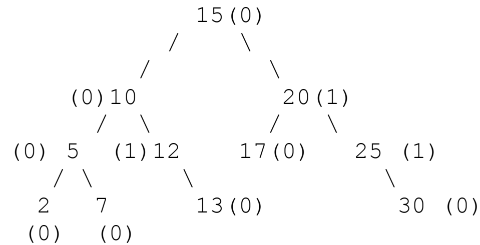
a situación antes da rotación sería:



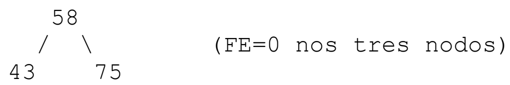
daquela o resultado é:



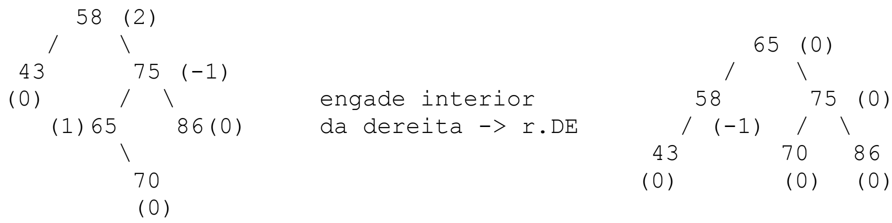
e cos FE's:



Dada a árbore inicial



insertar 86, 65, 70, 67, 73, 93, 69, 25, 66, 68, 47, 62, 10, 60



(sen acabar)

```
{═══════════════════════════════════════════════════════════════════════════}
procedure inserta_balanceada
( var nodo : tiponodo ; var bo : boolean ; info : tipoinfo ) ;
{
     nodo é a árbore AVL , info é a información a insertar

     Inicialmente bo é FALSE
}
var
   outro , nodo1 , nodo2 : tiponodo ;
begin
     if ( nodo <> nil ) then begin
       if ( info < nodo ^ . info ) then begin
         insertar_balanceada ( nodo ^ . esquerda , bo , info ) ;
         if ( bo ) then begin
           case nodo ^ . fe of
           1 : begin
                    nodo ^ . fe := 0 ;
                    {
                    se FE se fai cero bo é false porque nos nodos
                    anteriores non se incrementan niveis:
                            o
                          /   \         x é o nodo insertado
                         o     o
                        / \   / \       o FE do pai de x pasa de 1 a 0
                       x   o o   o
                    }
                    bo := false ;
               end ; { o factor de equilibrio é 1 }
           0 : nodo ^ . fe := -1 ; { estaba equilibrado e se engade á esquerda
}
           -1 : begin { hai que reestructurar a árbore }
                     nodo1 := nodo ^ . esquerda ;
                     if ( nodo1 ^ . FE = -1 ) then begin
                       {--------- rotación EE ----------}
                       nodo ^ . esquerda := nodo1 ^ . dereita ;
                       nodo1 ^ . dereita := nodo ;
                       nodo ^ . fe := 0 ;
                       nodo := nodo1 ;
                     end { o fillo esquerdo do nodo tén FE = -1 }
                     else begin { o fillo esquerdo do nodo NON tén FE -1 }
                         {--------- rotación ED ----------}
                         nodo2 := nodo1 ^ . dereita ;
                         nodo ^ . esquerda := nodo2 ^ . dereita ;
                         nodo2 ^ . dereita := nodo ;
                         nodo1 ^ . dereita := nodo2 ^ . esquerda ;
                         nodo2 ^ . esquerda := nodo1 ;
                         if ( nodo2 ^ . fe = -1 )
                         then nodo ^ . fe := 1
                         else nodo ^ . fe := 0 ;
                         if ( nodo2 ^ . fe = 1 )
                         then nodo1 ^ . fe := -1
                         else nodo1 ^ . fe := 0 ;
                         nodo := nodo2 ;
                     end ; { o fillo esquerdo do nodo NON tén FE -1 }
                end ; { fin de : o factor de equilibrio é -1 }
```

```
           end ; { case }
           nodo ^ . fe := 0 ;
           bo := false
         end ; { bo é TRUE }
       end { fin de : a información a insertar é menor que a da raiz }
       else
           if ( info > nodo ^ . info ) then begin
             inserta_balanceada ( nodo ^ . dereita , bo , info ) ;
               if ( bo ) then begin { bo é TRUE }
                 case nodo ^ . fe of
                 -1 : begin
                           nodo ^ . fe := 0 ;
                           bo := false ;
                      end ; { nodo ^ . fe = -1 }
                 0 : nodo ^ . fe := 1 ;
                 1 : begin { hai que reestructurar a árbore }
                          nodo1 := nodo ^ . dereita ;
                          if ( nodo1 ^ . fe = 1 ) then begin
                            {--------- rotación DD ---------}
                            nodo ^ . dereita := nodo ^ . esquerda ;
                            nodo1 ^ . esquerda  := nodo ;
                            nodo ^ . fe := 0 ;
                            nodo := nodo1 ;
                          end ; { nodo1 ^ . fe = 1 }
                     end { nodo ^ . fe = -1 }
                     else begin { nodo ^ . fe <> -1 }
                         {--------- rotación DE ---------}
                         nodo2 := nodo1 ^ . esquerda ;
                         nodo ^ . dereita := nodo2 ^ . esquerda ;
                         nodo2 ^ . esquerda := nodo ;
                         nodo1 ^ . esquerda := nodo2 ^ . dereita ;
                         nodo2 ^ . dereita := nodo1 ;
                         if ( nodo2 ^. fe = 1 )
                         then nodo ^. fe := -1
                         else nodo ^. fe := 0 ;
                         if ( nodo2 ^ . fe = -1 )
                         then nodo1 ^ . fe := 1
                         else nodo1 ^ . fe := 0 ;
                         nodo := nodo2 ;
                     end ; { nodo ^ . fe <> -1 }
               end ; { bo é TRUE }
             end ; { case }
             nodo ^ . fe := 0 ;
             bo := false ;
           end ; { a información a insertar é maior que a da raiz }
     end { fin de : se a árbore NON está inicialmente baleira }
     else begin { a árbore está inicialmente baleira }
         new ( outro ) ;
         outro ^ . info := info ;
         outro ^ . esquerdo := nil ;
         outro ^ . dereito := nil ;
         outro ^ . fe := 0 ;
         bo := true ;
         nodo := outro ;
     end ; { fin de se a árbore SE está inicialmente baleira }
end ; { inserta_balanceada }
{────────────────────────────────────────────────────────────────────────────}
```

Supresión en AVL

Consiste en quitar un nodo da árbore sen violar os principios que definen unha árbore equilibrada.

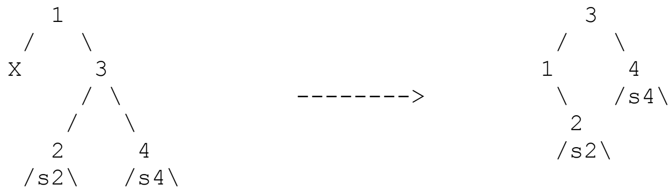
PRIMEIRO PASO:

Casos:
-se o elemento a suprimir é unha folla se suprime e iso é todo.
-se o elemento a suprimir tén un so descendente tén que substituirse por ese descendente
-se o elemento a suprimir tén dous descendentes, tén que substituirse polo nodo que se atopa máis á esquerda na subárbore dereita, ou polo nodo que se atopa máis á dereita na subárbore esquerda.

SEGUNDO PASO:

Reequilibrar se é necesario.

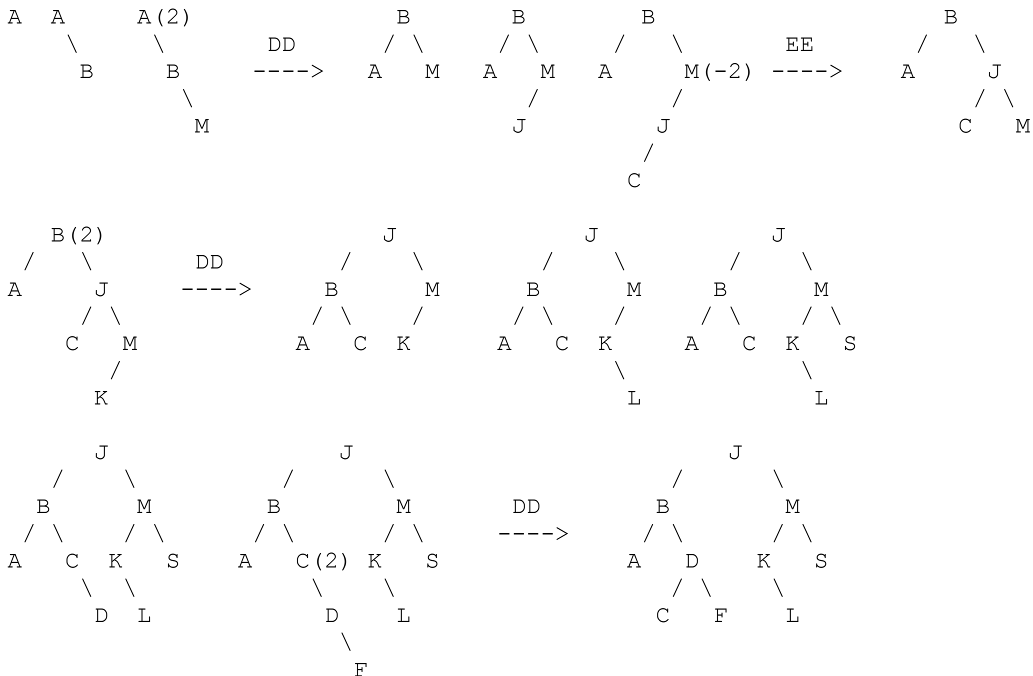
Hai un caso que nunca se daba engadindo nodos (é un "quinto caso de rotación"):

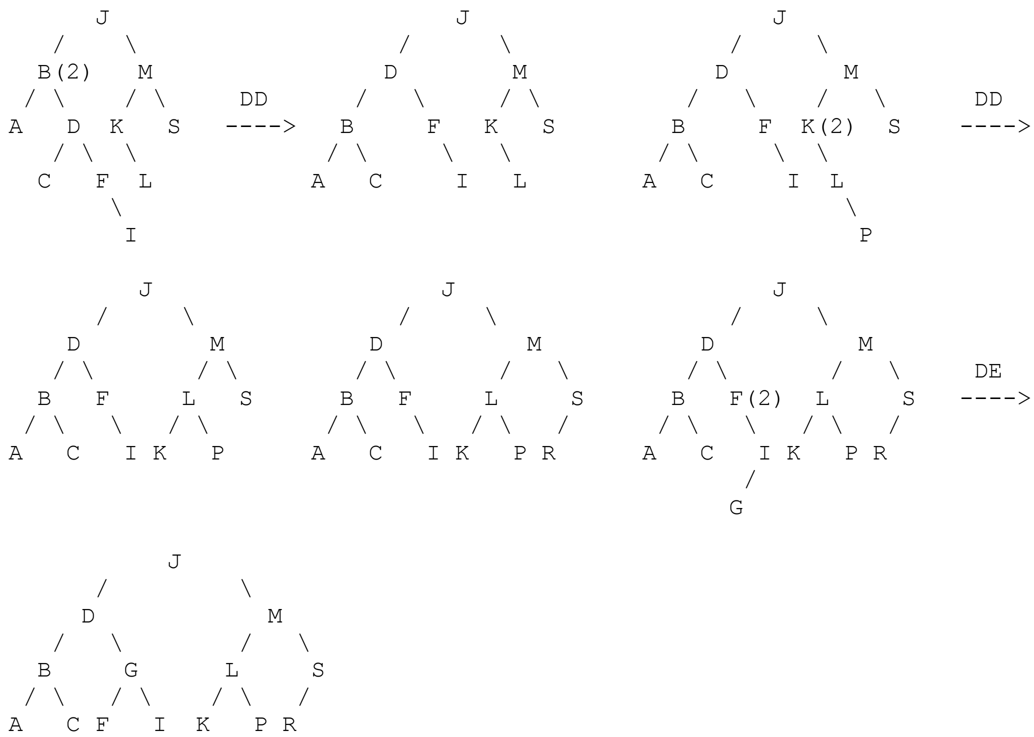


-buscar o Cairó a errata que hai no algoritmo de supresión.

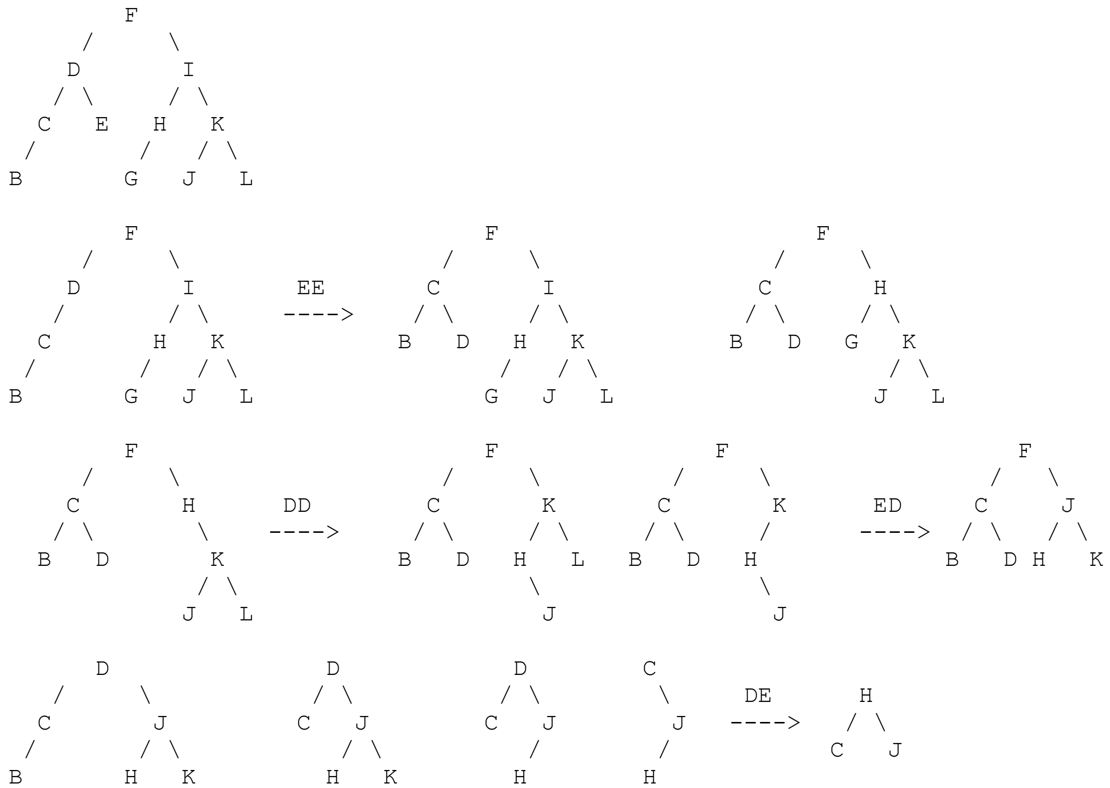
EXERCICIOS RESOLTOS.-

exercicio: secuencia de creación A, B, M, J, C, K, L, S, D, F, I, P, R, G

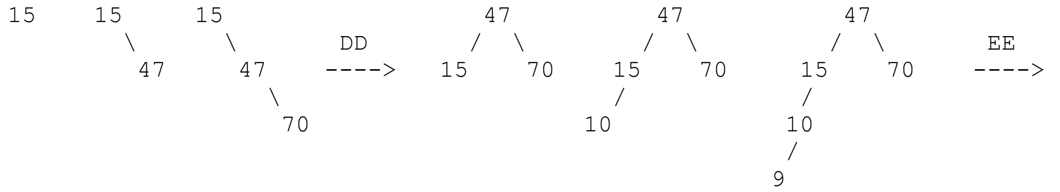


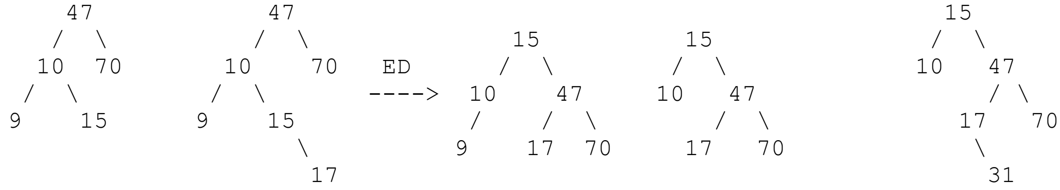


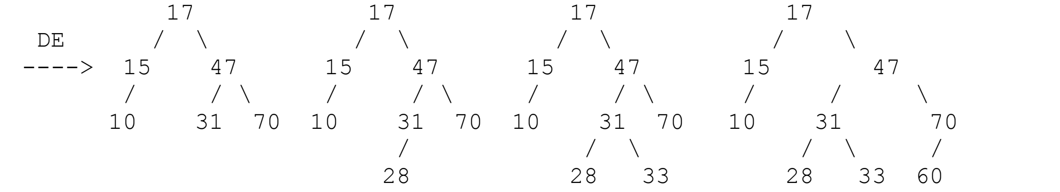
exercicio: secuencia de supresión E, I, G, L, F, B, K, D

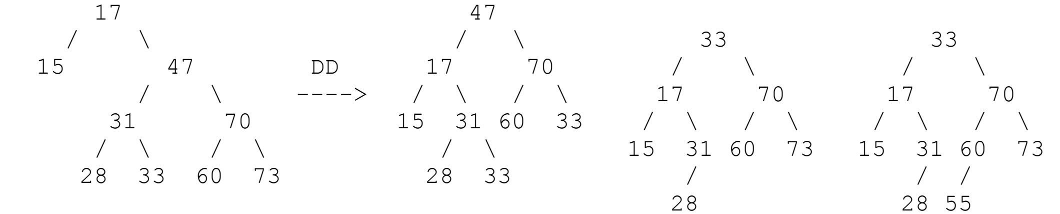


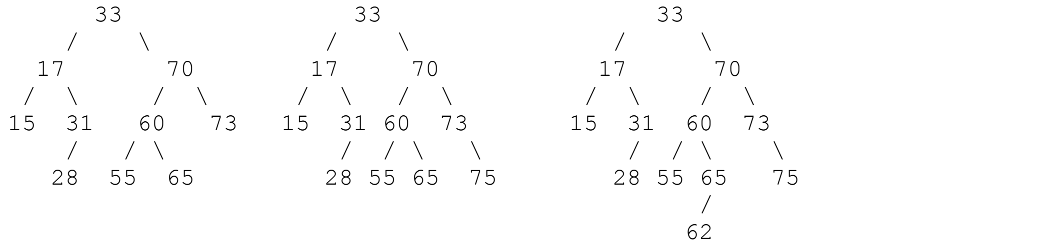
exercicio: + entran; - saen; +15, +47, +70, +10, +9, +17, -9, +31, +28, +33, +60, +73, -10, -47, +55, +65, +75, +62, -33.

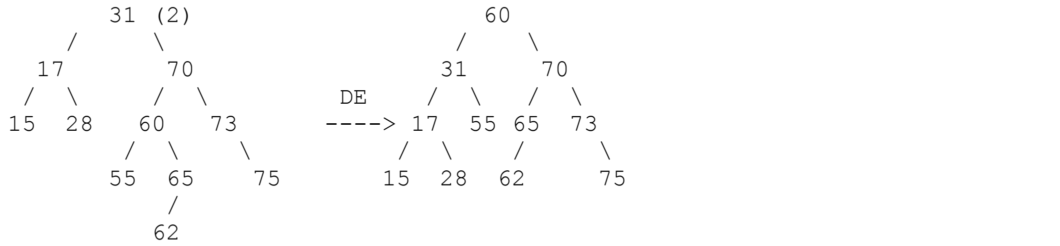






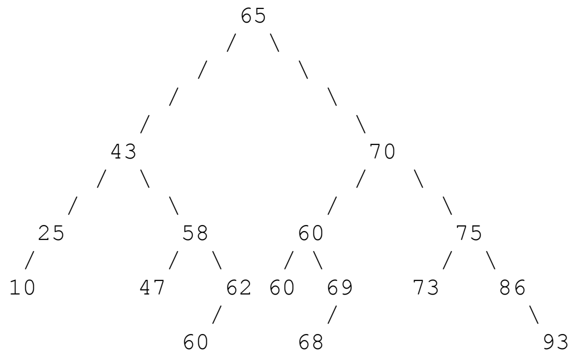






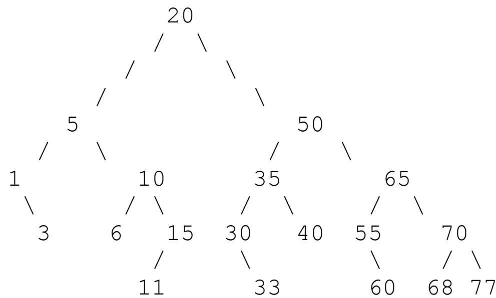
EXERCICIOS PROPOSTOS.-

exercicio: secuencia de supresión: 25, 75, 66, 65, 62, 10, 43, 47
na árbore seguinte



exercicio: secuencia de construcción: 50, 35, 75, 1, 7, 15, 40, 9, 80, 36, 20, 30, 45, 8, 42, 38, 65, 48, 41

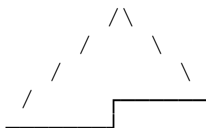
exercicio: secuencia de supresión: 55, 35, 50, 33, 40, 60, 65, 6, 5
na árbore seguinte:



MONTICULOS

Un montículo é unha árbore binaria que satisfai dúas propiedades: unha concerninte á súa figura e outra á orde dos seus elementos:

-é unha _árbore binaria_ que está _completa_ ou _case completa_, cas follas do derradeiro nivel tan á esquerda como sexa posible



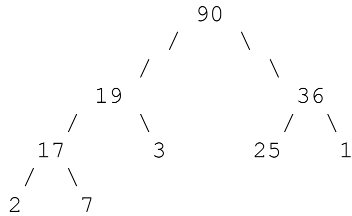
-para todo nodo, cómprese que o seu valor é maior ou igual que o valor de calquera dos seus fillos.

Empréganse para:
-operacións de ordenación -melloran a ordenación por mistura, complexidade n log 2 (n)-.
-ás veces para o manexo de colas de prioridade.

Representación dun montículo nun vector lineal.

Hai que lembrar que:
-o nodo k almacénase na posición k correspondente do arreglo
-o fillo esquerdo do nodo k almacénase na posición 2¨k
-o fillo dereito do nodo k almacénase na posición 2¨k+1

exemplo: dado o montículo...



...a representación nun vector lineal é:

90 19 36 17 3 25 1 2 7

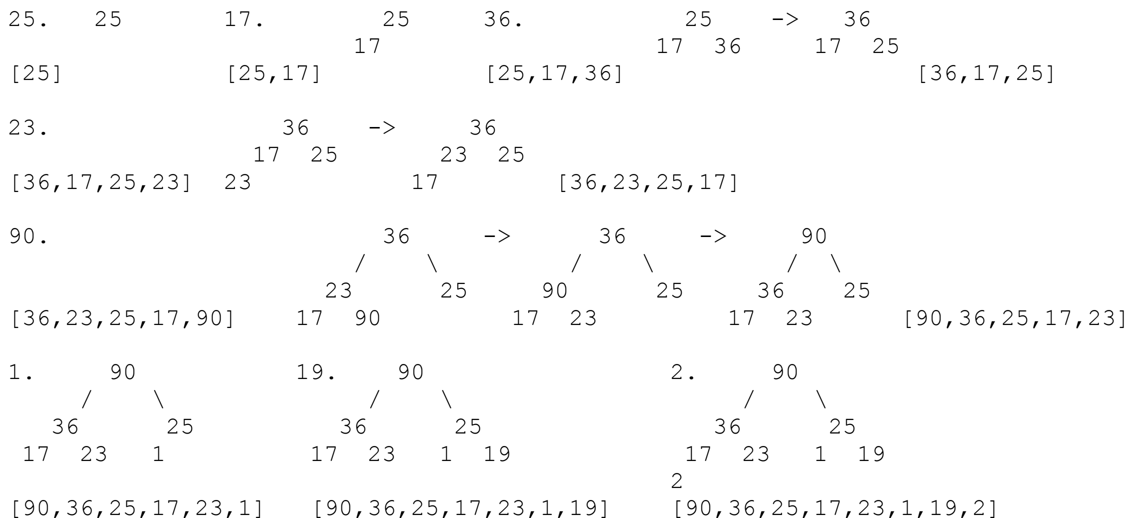
(é un percorrido de anchura do montículo)

```
anchura ( a )
     limpacola ( cola )
     if a <> nil
     then metecola ( a , cola )
     while not colabaleira ( cola ) do begin
          sacacola ( a )
          procesar ( a )
          if a ^ . esq <> nil
          then metecola ( a ^ . esq , cola )
          if a ^ . der <> nil
          then metecola ( a ^ . der , cola ) ;
```

Inserción en montículo.

A operación de inserción consta de dous pasos:
-insértase o elemento na primeira posición dispoñible no vector.
-verifícase se o seu valor é maior que o do seu pai. Se é así intercámbianse entre sí estes dous valores. Se non é así, o algoritmo chegou á súa fin. Este paso aplícase de forma recursiva de abaixo cara arriba.

exemplo: crear un montículo dada a secuencia 25 17 36 23 90 1 19 2



Ordenación por montículos ou heapsort.

E' o máis eficiente dos métodos de ordenación que usan árbores.
Consta de dous pasos: construir un montículo; e sacar os elementos do mesmo, pola raiz e segundo un algoritmo que veremos.

Algoritmo de transformación dun array desordeado nun montículo.

Supoñemos un arreglo con elementos de 1 a n, datos desordenados de tipo enteiro. Pasándolle o arreglo, o procedemento troca os valores de sitio de modo que o array resultante represente un montículo.

```
monticulo ( a : vector ; n : enteiros )
variables
         i , k , aux : enteiros
         b : booleana
comezo
          i inicialízase a 1, o comezo do vector
          mentres i <= n facer
               k <- i
               b <- certo
               mentres k > 1 e b = certo facer
                    b <- falso
                    se a [ k ] > a [ enteiro k/2 ] entón
                         trocar ( a [ k ] , a [ enteiro k/2 ] )
                         k <- enteiro k/2
                         b <- certo
                    finse
               finmentres

               i <- i+1

          finmentres
fin
```

Implementación dinámica.

Para engadir un elemento faise un percorrido en anchura e insértase no primeiro nodo que teña un fillo a nil.
Poderáse engadir un campo punteiro ó pai do nodo e poderáse usar recursividade.

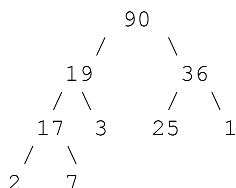
Eliminación dun montículo.

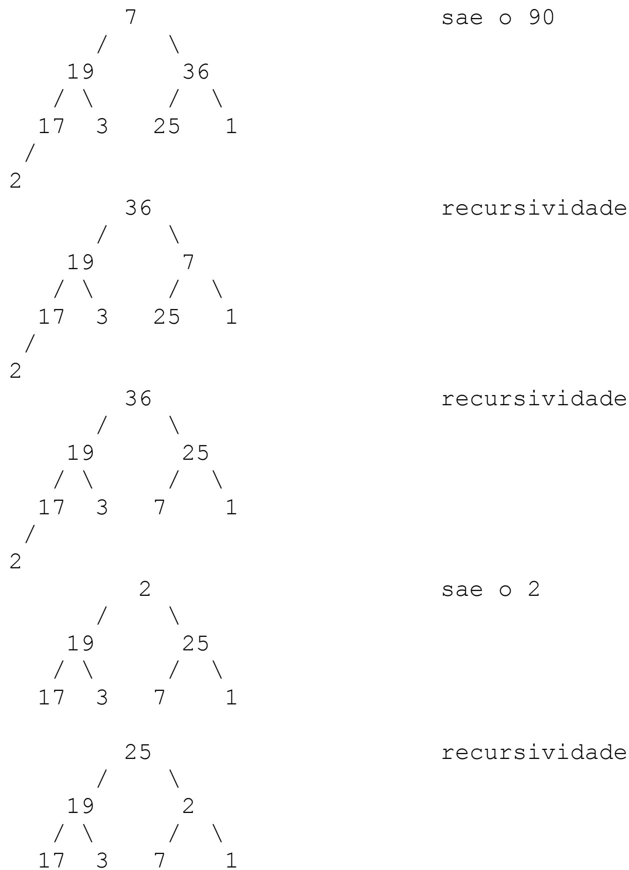
O proceso para obtener os elementos ordenados efectuaráse da seguinte maneira:

1º) reemprázase a raiz co elemento que ocupa a derradeira posición do montículo.

2º) verifícase se o valor da raiz é maior cós seus dous fillos (maior có maior dos seus fillos). Se é así, remata o algoritmo. Se non é así, trócase a raiz co maior. Aplícase de forma recursiva dende arriba cara abajo.

exemplo: [90,19,36,17,3,25,1,2,7]





```
        7                     sae o 90
      /    \
    19      36
   / \     /  \
  17  3   25   1
 /
2
        36                    recursividade
      /    \
    19      7
   / \     /  \
  17  3   25   1
 /
2
        36                    recursividade
      /    \
    19      25
   / \     /  \
  17  3   7    1
 /
2
         2                    sae o 2
      /    \
    19      25
   / \     /  \
  17  3   7    1

        25                    recursividade
      /    \
    19      2
   / \     /  \
  17  3   7    1
```

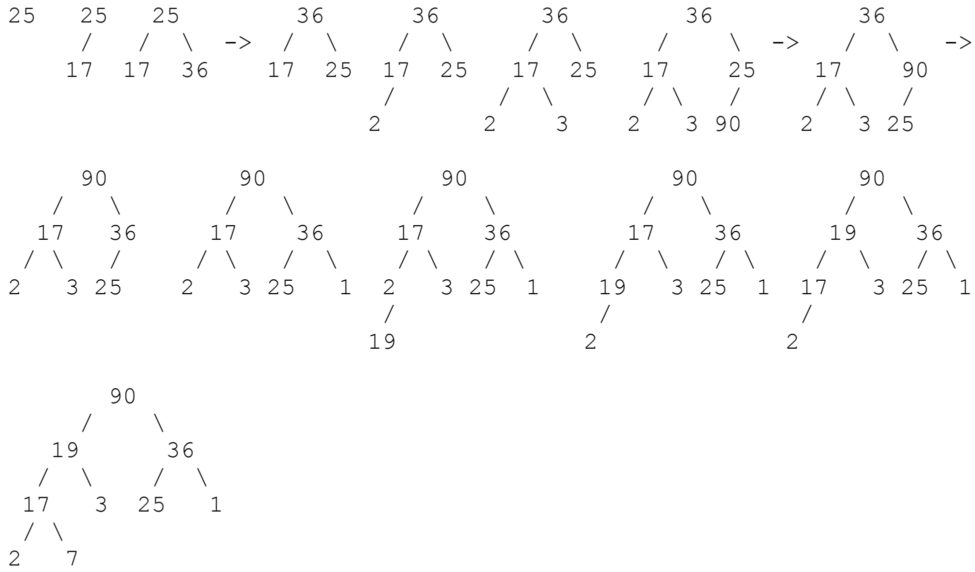
...etc

Os montículos son estructuras que serven para representar colas de prioridade.

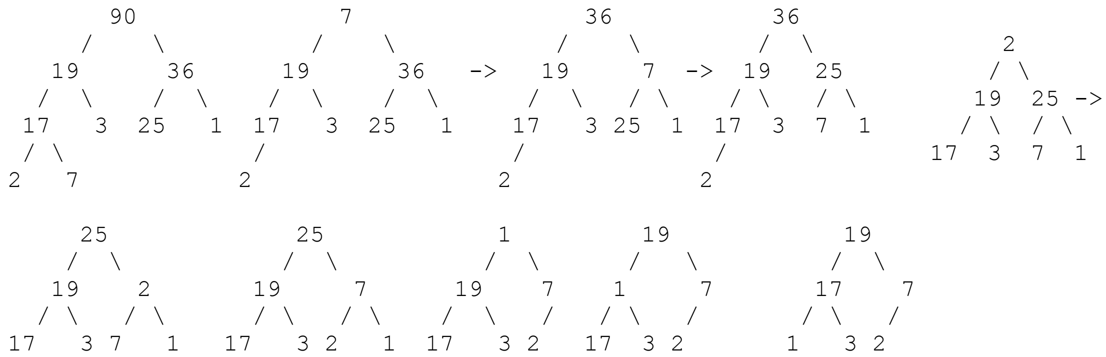
```
eliminar ( a : vector ; n : enteiro ) ;
variables
          i , aux , z , d , k , ad , maior : enteiros
comezo
          i <- n
          mentres i > 1 facer
               aux <- a[i]
               a[i] <- a[1]
               z <- 2
               d <- 3
               k <- 1
               mentres z < i facer
                    maior <- a[z]
                    ad <- z
                    se maior < a[d] e d <> i entón
                         maior <- a[d]
                         ad <- d
                    se aux < maior entón
                         a[k] <- a[ad]
                    k <- ad
                    z <- k¨2
                    d <- z+1
               finmentres
               a[k] <- aux
               i <- i-1
          finmentres
fin
```

EXERCICIOS RESOLTOS.-

exercicio: crea montículo 25, 17, 36, 2, 3, 90, 1, 19, 7



exercicio: quitar tres elementos do seguinte montículo.



EXERCICIOS PROPOSTOS.-

pasando a raiz da seguinte estructura e aproveitando as características de árbore enhebrada, sen recursividade nin pilas, con nodo cabeceira, facer o percorrido preorde e inorde.