Arbres binaires

Structure de données :

①

```
type  arbre = ^noeud ;
      noeud = record
              | val : element ;
              | g,d : arbre
              end ;
```

Parcours :

```
procedure prefixe (a:arbre) ;
Begin
  if (a<>nil) then
    Begin
      traiter(a) ;
      prefixe(a^.g) ;
      prefixe (a^.d) ;
    end ;
end ;
```

(RGD)

```
procedure infixe (a : arbre) ;
Begin
  if (a<>nil) then
    Begin
      infixe (a^.g) ;
      traiter (a) ;
      infixe (a^.d) ;
    end ;
end ;
```

(GRD)

```
Procedure postfixe (a : arbre) ;
Begin
  if (a<>nil) then
    Begin
      Postfixe (a^.g) ;
      Postfixe (a^.d) ;
      traiter (a) ;
    end ;
end ;
```

(GDR)

taille d'un arbre binaire

= Nombre de noeuds.

$$taille(A) = \begin{cases} 0 & \text{si } A \text{ est vide} \\ 1 + taille(g) + taille(d) & \text{sinon.} \end{cases}$$

(2)

```
function taille (a: arbre): integer;
Begin
    if (a = nil) then taille := 0
    else taille := 1 + taille(a^.g) + taille(a^.d);
end;
```

Nombre de feuilles d'un arbre binaire

$$nbfeuille(a) = \begin{cases} 0 & \text{si } A \text{ est vide} \\ 1 & \text{si } A \text{ est une feuille} \\ nbfeuille(g) + nbfeuille(d) & \text{sinon} \end{cases}$$

```
function nbfeuille(a: arbre): integer;
Begin
    if (a = nil) then nbfeuille := 0
    else if (a^.g = nil) and (a^.d = nil)
        then nbfeuille := 1
        else nbfeuille := nbfeuille(a^.g) +
                          nbfeuille(a^.d);
end;
```

tester si un noeud est une feuille.

feuille si gauche et droit = nil.

```
function feuille (a: arbre): boolean;
 Begin
   if (a = nil) then feuille := false
   else feuille := (a^.g = nil) and (a^.d = nil);
 end;
```

Nombre de feuilles (Procédure).

```
Procedure nbfeuille (a: arbre; var nb: integer)
Begin
   if (a <> nil) then
   Begin
      if (feuille (a)) then nb := nb+1;
      nbfeuille (a^.g);
      nbfeuille (a^.d);
   end;
end;
```

Rechercher un élément :

```
function appartient (val x : element ; a : arbre) : boolean;
Begin
    if (a = nil) then appartient := false
    else if (a^.val = x) then appartient := true
         else appartient := appartient (x, a^.g)
                            or appartient (x, a^.d);
end;
```

hauteur d'un arbre binaire

= nombre de niveaux.

```
function hauteur (a : arbre) : integer;
var h1, h2 : integer;
Begin
    if (a = nil) then hauteur := -1
    else Begin
            h1 := hauteur (a^.g);
            h2 := hauteur (a^.d);
            if h1 > h2 then hauteur := h1 + 1
            else hauteur := h2 + 1;
         end;
end;
```

(4)

hauteur d'un noeud = niveau.

```
function hauteur (a: arbre ; nd: elt): integer;

    function haut (a: arbre): integer;
    Var h: integer;
    Begin
      if (a=nil) then haut := -1
      else if a^.val = nd then haut := 0
           else Begin
                  h := haut (a^.g);
                  if h = -1 then h := haut(a^.d);
                  if h = -1 then haut := -1
                            else haut := h+1;
                end;
    end.

Begin
  hauteur := haut (a);
end;
```

5

Nombre de descendants (liste).

```
type  ptr = ^ elt;
      elt = record
              val : element;
              suiv : ptr;
            end.
```

} liste dans laquelle mettre les descendants.

```
function descendant(a: arbre; nd: element) : ptr;
      function des (a: arbre) : ptr;
      Begin
        if (a=nil) then des := nil
        else if a^.val = nd then
                               Begin
                                 par (a^.g);
                                 par (a^.d);
                                 des := tete;
                               end.

             else Begin
                    tete := des (a^.g);
                    if tete = nil then
                      des := des (a^.d);
                    des := tete;
                  end.
      end.
  Begin
      descandant := des (a);
  end.
```

(7)

```
Procedure par (a : arbre);
var p : ptr;
Begin
    if (a <> nil) then
        Begin
            new (p);
            p^.val := a^.val;
            p^.suiv := tete;
            tete := p;
            par (a^.g);
            Par (a^.d);
        end;
end;
```

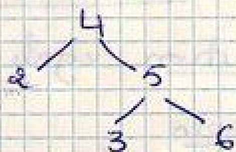
(8)

Ascendants (liste).

```
function ascendants (a : arbre; nd : element) : ptr;
var p, tete : ptr; b : arbre;
Begin
    tete := nil; b := pere (a, x);
    while (b <> nil) do
        Begin   new(p); p^.val := b^.val;
                        p^.suiv := tete;
                        tete := p;
                x := b^.val; b := pere (a, x);
        end;
end. ascendant := tete;
```

Arbre binaire de recherche.

gauche < racine
droit > racine



Chercher un noeud

```
function appartient (a: arbre ; nd: elt): boolean.
Begin
    if (a = nil) then appartient := false
    else if (a^.val = nd) then appartient := true
         else
            if (nd < a^.val)
            then appartient := appartient (a^.g, nd)
            else appartient := appartient (a^.d, nd).
end.
```

(9)

Ajouter un noeud

```
procedure ajout (var a: arbre ; x: element).
Begin
    if (a = nil) then Begin
                        new(a);
                        a^.val := x.
                        a^.g := nil.
                        a^.d := nil.
                      end
    else    if x > a^.val then ajout (a^.g, x)
            else ajout (a^.d, x).
end.
```

Tester l'égalité de deux arbres

```
function egale (a1, a2 : arbre) : boolean.

Begin
   if (a1 = nil) and (a2 = nil) then
         egale := true
   else if (a1 = nil) or (a2 = nil) then
         egale := false
   else egale := (a1^.val = a2^.val) and
                 egale (a1^.g, a2^.g) and
                 egale (a1^.d, a2^.d).
end,
```

10

Arbres planaires

Structure de données

```
type   arbre = ^noeud;
            ptr = ^elt;
            elt = record
                    elem: element;
                    suiv: ptr;
                  end;
(1)
       noeud = record
                 val: element;
                 fils: ptr;
                 suiv: arbre;
               end;
```

Chercher un noeud

```
function appartient(a: arbre; nd: element): boolean;
  var  tr: boolean;
  Begin
      tr := false;
      while (a <> nil) and (not (tr)) do
      if a^.val = nd then tr := true else
        a := a^.suiv;

      appartient := tr;
  end;
```

Trouver le père d'un noeud.

```
function pere (a: arbre; nd: element) : arbre;
var tr: boolean;
Begin
    tr:= false;
    if (a^.val = nd) then  pere:= nil
    else Begin
        while (a<>nil) and (not(tr)) do
        if appartient (a^.fils, nd) then
            tr:= true  else  a:= a^.suiv.
        pere:= a;
    end.
end.
```

(2)

hauteur d'un noeud

```
function hauteurnd(a: arbre; nd: elt): integer;
var c: integer;
Begin
    c:= 0;
    while (pere (a, nd) <> nil) do Begin
                                       c:= c + 1;
                                       nd:= pere (a, nd)^.val.
                                   end;
    hauteurnd:= c;
end.
```

Tester si un noeud est une feuille

```
function feuille (a: arbre): boolean;
Begin
   if (a = nil) then feuille := false
   else feuille := a^.fils = nil;
end;
```

Calculer le nombre de feuilles

```
function nb_feuille (a: arbre): integer;
  var n: integer;
Begin
   n := 0;
   while (a <> nil) do
       Begin
          if (a^.fils = nil) then n := n + 1;
          a := a^.suiv;
       end;
   nb_feuille := n;
end;
```

(3)

Descendants d'un noeud.

```
function descendant (a : arbre ; nd : element) : ptr ;
var  tete : ptr ;
      procedure des (p : ptr) ;
       var  q : ptr ;
      Begin
          while (p<>nil) do
                 Begin
                     new(q) ;
                     q^.val := p^.val ;
                     q^.suiv := tete ;
                     tete := q ;
                     des (recherche (a, p^.val)^.fils) ;
                     p := p^.suiv ;
                 end ;
      end ;
  Begin
      tete := nil ;
      des ( rechere (a, x)^.fils) ;
      descendant := tete ;
  end.
```

5

```
        function recherche (a : arbre ; nd : elt) : ptr ;
        Begin
             While (a<>nil) and (a^.val <> nd) do
                   a := a^.suiv ;
             recherche := a^.fils ;
        end.
```