# 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);
      end. traiter(a);
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): booleen
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;
```

hauteur d'un nœud = 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 ;
```

## Longeur de cheminement

= Somme des hauteurs

```
function LC (a: arbre) : integer;

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

Begin
  if (a = nil) then LC := -1
  else LC := long (a, 0);
end;
```

# 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;
```

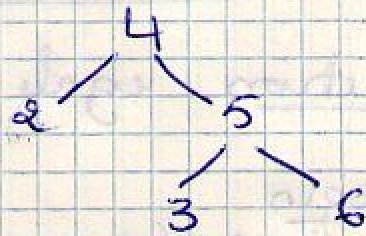
8

## 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 nœud

```
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 nœud

```
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

Parcours en largeur (par niveaux) d'un arbre binaire

```
Procedure largeur (a : arbre);
  var  f : file;   x : arbre;
  Begin
     init_file (f);
     if (a <> nil) then  ajouter (f, a);
     while not (file_vide (f)) do
        Begin
           x := premier (f);
           retirer (f);
           traiter (x);
           if (a^.g <> nil) then ajouter (f, a^.g);
           if (a^.d <> nil) then ajouter (f, a^.d);
        end;
  end;
```

## Supprimer un noeud

(12)

```
procedure supp (a: arbre);
 var p, q: arbre;
Begin
    p:=a;
    if (a^.d <> nil) then
        Begin
            q:= p^.d;
            while (q^.g <> nil) do q:= q^.g;
            q^.g := p^.g;
            a:= a^.d;
        end
    else a:= a^.g;
    dispose (p);
end;
```

(itérative)

```
procedure supprime (var a: arbre; nd: element).
Begin
    if (a <> nil) then
      if a^.val = nd then supp (a)
    else if nd < a^.val then supprime (a^.g, nd)
        else supprime (a^.d, nd);
end.
```

(récursive)

```
function pere (a: arbre ; nd: element): arbre;
 var trouve: boolean;
Begin
    trouve := false;
   if (a = nil) then pere := nil
   else Begin
          if (a^.g^.val = nd) or (a^.d^.val = nd)
          then Begin
                    pere := a;
                    trouve := true;
                end;
       else if not(trouve) then
              Begin
                    pere := pere(a^.g, nd);
                    pere := pere(a^.d, nd);
                end;
end;
```

Fonction qui donne le père d'un noeud donné

```
Procedure vider (a: arbre);
Begin
    if (a <> nil) then
      Begin
        if (a^.g <> nil) then vider (a^.g);
        if (a^.d <> nil) then vider (a^.d);

        dispose (a);
      end;
end;
```

14

Vider un arbre binaire