Chapitre 16 – Les arbres binaires

Structure arborescente.

Un arbre est un ensemble d'éléments appelés nœuds (dont un se singularise comme la racine) liés par une relation hiérarchique. Un nœud, comme tout élément d'une liste, peut-être de n'importe quel type.

Définition : arbre

- Un nœud unique est un arbre. Il est la racine de l'arbre.
- Si n est un noeud et A_1, ..., A_k sont des arbres de racine respective n_1, ..., n_k, on peut construire un nouvel arbre en associant comme parent unique aux noeuds n_1 , ..., n_k le noeud n. Dans cet arbre, n est la racine et A_1, ..., A_k sont les sous-arbres de cette racine. Les noeuds n_1, ..., n_k sont appelés les fils de n.

L'arbre vide (noté Λ) est un arbre sans noeud.

Définition : arbre binaire (1/2)

- Un arbre binaire est soit un arbre vide, soit un arbre où chaque noeud a un fils gauche, un fils droit ou les 2.
- $B = \Lambda + <o, B_1, B_2>$ où B_1 et B_2 sont des arbres binaires disjoints et 'o' est un noeud appelé racine. B_1 est le sous-arbre gauche et B_2 est le sous-arbre droit.

  Attention ! Non symétrie gauche/droite des arbres binaires.
- On dit que C est un sous-arbre de B ssi C=B ou $C=B_1$ ou $C=B_2$ ou C est un sous-arbre de B_1 ou de B_2.
- Si un noeud n_i a pour fils gauche (ou droit) un noeud n_j, n_i est le père de n_j.
- Deux noeuds qui ont le même père sont dits frères.
- Le noeud n_i est un ascendant du noeud n_j ssi n_i est le père de n_j, ou un ascendant du père de n_j ; et n_i est un descendant de n_j ssi n_i est fils de n_j, ou n_i est un descendant d'un fils de n_j.

Définition : arbre binaire (2/2)

- Un noeud sans fils est appelé feuille. Les autres sont des noeuds internes.
- Une branche de l'arbre est tout chemin (suite de noeuds consécutifs) de la racine à une feuille. Un arbre a autant de branches que de feuilles.
- La **hauteur** d'un noeud (ou profondeur) est définie récursivement par :
  soit x noeud de B, h(x) = 0 si x est la racine de B
  h(x) = 1+ h(y) si y est le père de x.
- La hauteur d'un arbre B est h(B) = 0 si B est vide
  max {h(x), x noeud de B}
- Un arbre binaire est **entier** ssi tous ses nœuds possèdent zéro ou 2 fils.
- Un arbre binaire est **complet** ssi ses feuilles ont pour profondeur *n* ou *n-1*.
- Un arbre binaire est **parfait** ssi toutes ses feuilles sont à la même profondeur.

TAD ArbreBinaire

Sorte : ArbreBinaire

Utilise : Noeud, Booléen, Elément

Opérations :
Créer-arbre-vide() → ArbreBinaire
Créer(Noeud, ArbreBinaire, ArbreBinaire) → ArbreBinaire
Racine(ArbreBinaire) → Noeud
Père(Noeud, ArbreBinaire) → Nœud
Fils-gauche(ArbreBinaire) → ArbreBinaire
Fils-droit(ArbreBinaire) → ArbreBinaire
Arbre-vide(ArbreBinaire) → Booléen
Affecter-val(Noeud, Elément) → Noeud
Val(Nœud) → Elément

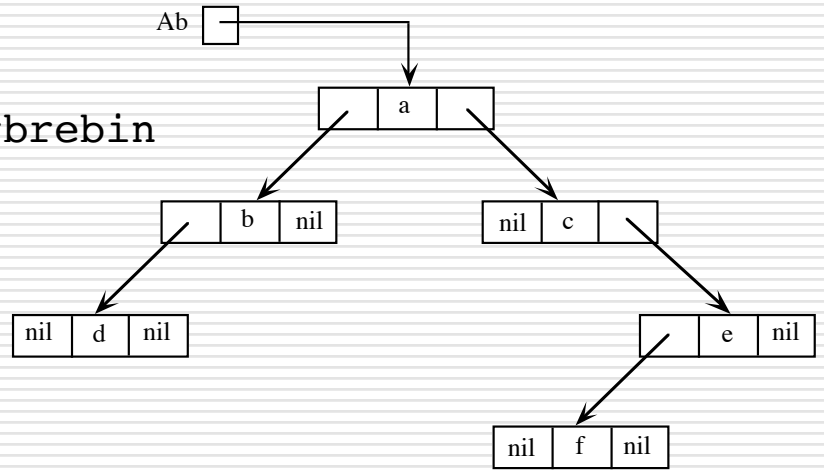
Implémentation du TAD

Par pointeurs

Chaque noeud contient deux pointeurs, l'un vers le sous-arbre gauche et l'autre vers le sous-arbre droit ; l'arbre est déterminé par l'adresse de sa racine.

```
Type    arbrebin = ^noeud
        noeud = Enregistrement
                   val : élément
                   gauche, droit : arbrebin
                FinEnregistrement
Var Ab : arbrebin

fils-gauche(Ab) = Ab^.gauche
fils-droit(Ab) = Ab^.droit
```



Implémentation du TAD

Par tableau

Les noeuds de l'arbre sont mis en bijection avec un ensemble d'adresses dans des « cases » gauche et droit d'un tableau correspondant aux racines des sous-arbres gauche et droit. L'adresse de la racine de l'arbre est contenue dans une variable Ab.

```
Type cellule = Enregistrement
                    val : élément
                    gauche, droit : entier
                FinEnregistrement
     tabarbre = tableau [1..n] de cellule
     arbrebin = entier
Var Ab : arbrebin
    t : tabarbre
Ab ← 4
fils-gauche(Ab) = t[Ab].gauche
fils-droit(Ab) = t[Ab].droit
```

Ab = 4

libre = 1 {Chaînage des cases libres dans un arbre}

t	val	gauche	droite
1	-	3	6
2	c	0	8
3	-	9	0
4	a	7	2
5	d	0	0
6	-	0	0
7	d	5	0
8	e	10	0
9	-	0	0
10	f	0	0

Parcours (récursif)

Profondeur à gauche : il suit le chemin qui part à gauche de la racine et va toujours le plus à gauche. Chaque noeud est rencontré 3 fois : à gauche, en dessous, à droite.

ordre préfixé (préordre) : seul traitement1 est appliqué

ordre infixé (symétrique) : seul traitement2 est appliqué

ordre sufixé (postfixé) : seul traitement3 est appliqué

```
Procédure profGauche (E A : Arbrebin)
Début
    Si arbre-vide(A) Alors terminaison
                     Sinon traitement1
                           profgauche(fils-gauche(A))
                           traitement2
                           profgauche(fils-droit(A))
                           traitement3
    FinSi
Fin
```

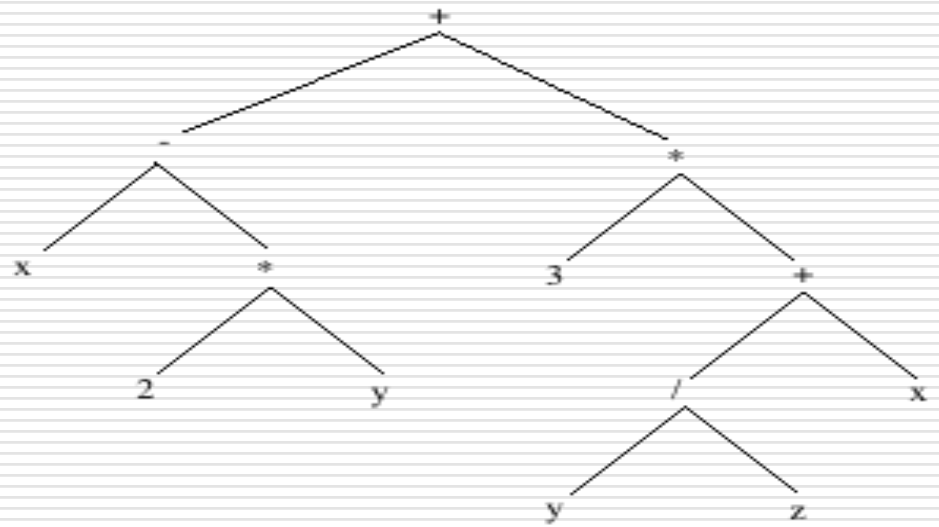
Exemple

Expression arithmétique x-2*y+3*(y/z+x)

préfixe : traitement1 = affiche(val(racine(A)) +-x*2y*3+/yzx

postfixe : traitement3 = affiche(val(racine(A)) x2y*-3yz/x+*+

infixe : traitement2 = affiche(val(racine(A)) x-2*y+3*y/z+x

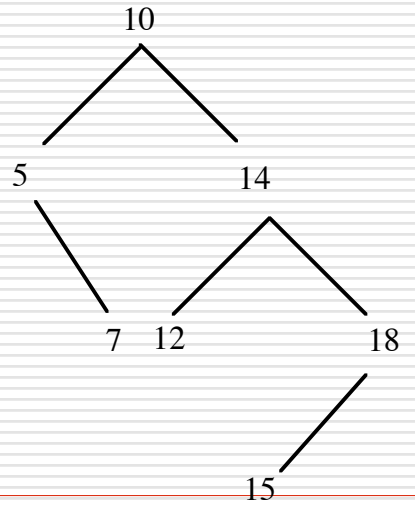


Problème de parenthèsage pour infixe

```
Procédure infixe (E A : arbrebin)
Début
Si arbre-vide(fils-gauche(A))
      et arbre-vide(fils-droit(A))  {feuille(A)}
   Alors affiche(val(racine(A)))
   Sinon affiche('(')
      infixe(fils-gauche(A))
      affiche(val(racine(A)))
      infixe(fil-droit(A))
      affiche(')')
FinSi
Fin
```

Arbre Binaire de Recherche (ABR)

- Propriété : dans un ABR, le sous-arbre gauche (resp. droit) du noeud x ne contient que des éléments de valeur inférieure (resp. supérieure) à celle de l'élément en x.
- Le test d'appartenance à l'ensemble des éléments de l'arbre est aisé.
- Pour l'insertion, on cherche à positionner x en attendant de rencontrer un noeud vide dans la descente pour le remplacer par un noeud contenant x.



Appartient

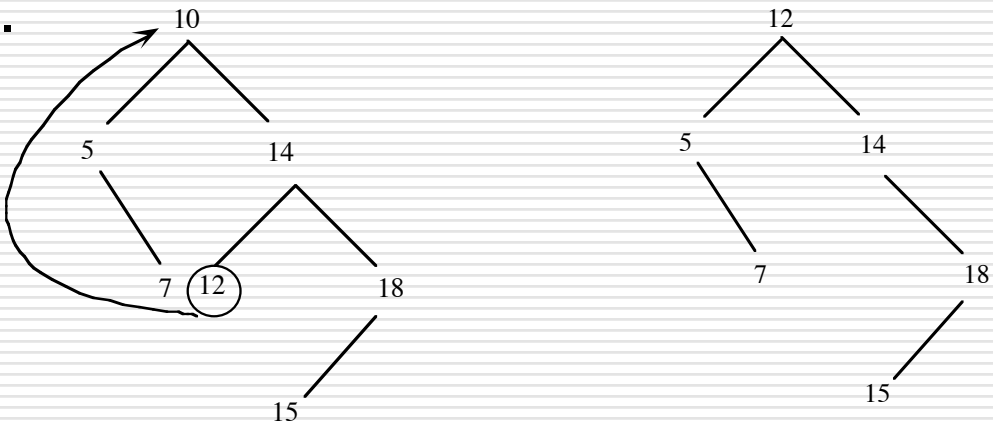
```
Fonction appartient (x : élément, A : arbrebin): booléen
Var app : booléen
Début
Si arbre-vide(A)
   Alors app ← faux
   Sinon Si x=val(racine(A))
           Alors app ← vrai
           Sinon Si x<val(racine(A))
                    Alors app ← appartient(x,fils-gauche(A))
                    Sinon app ← appartient(x,fils-droit(A))
                 FinSi
        FinSi
FinSi
retourne(app)
Fin
```

Insérer

```
Procédure insérer (E x : élément, E/S A : arbrebin)
Début
Si arbre-vide(A)
    Alors affecter-val(racine(A),x)
         fils-gauche(A) ← créer-arbre-vide
         fils-droit(A) ← créer-arbre-vide
    Sinon Si x<val(racine(A))
              Alors insérer(x,fils-gauche(A))
              Sinon Si x>val(racine(A))
                        Alors insérer(x,fils-droit(A))
                    FinSi
          FinSi
FinSi
Fin
```

Supprimer

- La suppression pose de petits problèmes.
- Il faut chercher le plus petit élément de l'arbre plus grand que l'élément à supprimer (ici, on supprime 10 donc la ppe est 12). Et on remplace l'élément à supprimer par le ppe.
- Besoin d'une procédure supprimin qui retire le ppe et retourne sa valeur.



Supprimin

```
Procédure supprimin (E/S A : arbrebin, S e : élément)
Début
Si arbre-vide(fils-gauche(A))
    Alors e ← val(racine(A))
          A ← fils-droit(A)
    Sinon supprimin(fils-gauche(A),e)
FinSi
Fin
```

```
Procédure supprimer (E x : élément, E/S A : arbrebin)
Début
Si ¬arbre-vide(A)
  Alors Si x<val(racine(A))
          Alors supprimer(x,fils-gauche(A))
          Sinon Si x>val(racine(A))
                  Alors supprimer(x,fils-droit(A))
                  Sinon Si arbre-vide(fils-gauche(A))
                          et arbre-vide(fils-droit(A))  {feuille(A)}
                          Alors A←créer-arbre-vide
                          Sinon Si arbre-vide(fils-gauche(A))
                                  Alors A←fils-droit(A)
                                  Sinon Si arbre-vide(fils-droit(A))
                                          Alors A←fils-gauche(A)
                                          Sinon supprimin(fils-droit(A),e)
                                                affecter-val(racine(A,e)
                                  FinSi
                          FinSi
                  FinSi
             FinSi
          FinSi
FinSi
Fin
```