UTBM le 14 janvier 2002

FINAL – LO42

Solution

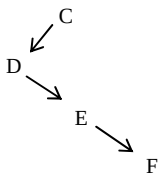
Les documents ne sont pas autorisés (La copie ou les idées du voisin non plus).
Toute réponse devra être claire et justifiée si nécessaire (toute ambiguïté sera mal interprétée).
L'élégance de la solution sera jugée. Vous éviterez les variables globales.
Sauf indication contraire, dans le cas d'algorithmes les réponses doivent être rédigées en pseudo code.
La première question sera faite sur de feuilles indépendantes aux deux autre questions.

I. L'arboriculture ! (8)

Quelques informations sur les arbres, notamment les arbres binaires :

- La taille d'un arbre est le nombre d'éléments contenus dans l'arbre.
- Un arbre binaire est dit dégénéré si sa topologie correspond à une liste.

Exemple :



a) Savons nous les regarder ?

On vous demande d'écrire une série de fonctions permettant de déterminer les propriétés suivantes :

- Taille d'un arbre binaire.

```
Fonction taille (A : arbreB) : entier ;
Debut
        Si A=Nul Alors taille ← 0 ;
        Sinon taille ← taille (A^.FG) + taille (A^.FD) + 1 ;
        Fsi ;
Fin ;
```

- Hauteur d'un arbre binaire,

```
Fonction hauteur (A : arbreB) : entier ;
Var     hg, hd : Entier ;
Debut
        Si A=Nul Alors hauteur ← 0 ;
        Sinon   hg ← hauteur (A^.FG) ;
                hd ← hauteur (A^.FD) ;
                Si hg > hd Alors hauteur ← hg + 1
                Sinon hauteur ← hd + 1
                Fsi ;
        Fsi ;
Fin ;
```

- Nombre de feuilles d'un arbre binaire,

```
Fonction nbfeuilles (A : arbreB) : entier ;
Var     nb : Entier ;
Debut
        Si A=Nul Alors nbfeuilles ← 0 ;
        Sinon   nb ← nbfeuilles (A^.FG) + nbfeuilles (A^.FD) ;
                Si nb>0 Alors nbfeuilles ← nb
                Sinon nbfeuilles ← 1
                Fsi ;
        Fsi ;
Fin ;
```

- Vérification qu'un arbre binaire n'est pas dégénéré,

Méthode 1 : Il est possible de vérifier le nombre de feuilles de l'arbre. On ajoutera un test pour les petits arbres que l'on choisit de considèrer comme normaux. Le test peut porter sur la hauteur ou la taille.

```
Fonction normal  (A : arbreB) : booléen ;
Debut
        Si A = Nul Alors normal ← vrai ;
        Sinon normal ← nbfeuilles (A) > 1ou hauteur(A)<=2 ;
Fin ;
```

Méthode 2 : Il suffit qu'un nœud révèle deux fils au cours d'un parcours en profondeur.

```
Fonction normal2  (A : arbreB) : booléen ;
Debut   Si A = nul Alors normal ← vrai ;
        Sinon   Si A^.FG <> nul Alors
                        Si A^.FD <> nul Alors normal ← vrai
                        Sinon normal ← normal (A^.FG) ;
                        Fsi ;
                Sinon normal ← normal (A^.FD) ;
                Fsi ;
        Fsi ;
Fin ;
```

- Vérification qu'un arbre binaire contenant des entiers est un ABR.

Méthode 1 : Un arbre A est un ABR si les sous arbres gauches et droits sont des ABR et la valeur contenue en A est supérieure au maximum du sous arbre gauche et inférieure au minimum du sous arbre droit.

```
Fonction ABRoupas  (A : arbreB) : booléen ;
Var     min,max : entier ;
Debut
        ABRoupas ←CQFD(A,min,max) ;
Fin ;

Fonction CQFD  (A : arbreB ;Var min, max : entier) : booléen ;
Var     babr : Booléen ;
        mind : Entier ;
Debut
        Si A = nul Alors CQFD ← vrai ;
        Sinon
                Si A^.FG=nul Alors
                        min ← A^.val ;
                        max ← min ;
                        babr ← vrai ;
                Sinon
                        babr ← CQFD(A^.FG, min, max) et A^.val >= max ;
                Fsi ;
                max ← A^.val ;
                Si min > A^.val Alors min ← A^.val ;
                Fsi ;
                Si babr Alors
                        mind ← A^.val ;
                        babr ← CQFD(A^.FD, mind, max) et A^.val <= mind ;
                Fsi ;
                CQFD ← babr ;
        Fsi
Fin ;
```

Méthode 2 : Un arbre A est un ABR si dans un parcours infixé la valeur de tout sommet est supérieure à toutes valeurs déjà vues.

```
Fonction ABRoupas2  (A : arbreB) : booléen ;
Var     max : entier ;
        test : booléen ;
Debut
        test ← faux ;
        ABRoupas2 ←CQFD2(A, max, test) ;
Fin ;

Fonction CQFD2  (A : arbreB ; Var max : entier ; Var test : Booléen ) : booléen ;
Var     ok : Booléen ;
Debut
        Si A = nul Alors CQFD2 ← vrai ;
        Sinon
              ok ← CQFD2  (A^.FG, max, test ) ;
              Si ok Alors
                  Si test Alors          /* pour initialiser au plus à gauche */
                        ok ← A^.val > max ;
                  Sinon test ← vrai ;
                  Fsi ;
                  max ← A^.val ;
```

```
                    CQFD2 ← ok et CQFD2  (A^.FD, max, test ) ;
                Sinon CQFD2 ← faux ;
                Fsi ;
        Fsi
Fin ;
```

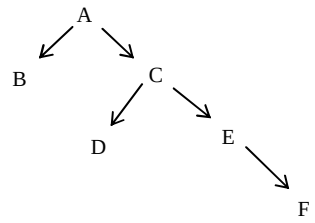
Méthode 3 (non souhaitée) : On parcourt l'arbre en ordre infixé et on remplit une pile. Si la pile est ordonnée en ordre décroissant l'arbre est ABR.

```
Fonction ABRoupas3  (A : arbreB) : booléen ;
Var     p : pile ;
        v1, v2  : entier ;
        ok : Booléen ;
Debut
        initialiser(f) ;
        parcours_et_pile (A, f) ;
        dépiler(p, v2)
        tantque Non pile_vide(p) et ok faire
                dépiler(p,v1) ;
                ok ← v1 <= v2 ;
                v2 ← v1 ;
        fait ;
        ABRoupas3  ← ok ;
Fin ;

Fonction parcours_et_pile  (A : arbreB ; Var p : pile ) : booléen ;
Debut
        Si A <> nul Alors
                parcours_et_pile  (A^.FG, p ) ;
                empiler(p, A^.val) ;
                parcours_et_pile  (A^.FD, p ) ;
        Fsi
Fin ;
```

L'utilisation d'autres structures de données n'est pas souhaitée.

b) Affichons leur anatomie

Nous avons l'habitude de représenter un arbre de façon graphique comme suit :



Nous pouvons représenter cet arbre sous forme textuelle comme suit :

A (B ((), ()), C (D ((), ()), E ((), F ((), ()))))

ou sous forme simplifiée

A (B, C (D, E (, F)))

Vous devez écrire une procédure permettant d'afficher un arbre binaire sous la forme complète.
Modifiez votre procédure pour afficher sous la forme réduite.

Le traitement est un parcours en profondeur de l'arbre où l'on intègre des traitements préfixés, infixés et postfixés.

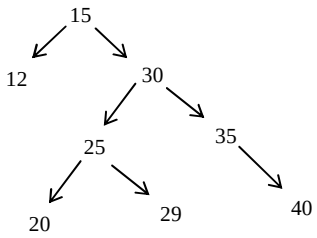
```
Fonction affiche1  (A : arbreB ) : booléen ;
Var     aux : arbreB ;
Debut   Si A = nul Alors Ecrire('()') ;
        Sinon   Ecrire( A^.val , '( ') ;
                affiche1 (A^.FG) ;
                Ecrire(', ') ;
                affiche1 ( A^.FD) ;
                Ecrire(' )') ;
        Fsi ;
Fin ;

Fonction affiche2  (A : arbreB ) : booléen ;
Var     aux : arbreB ;
Debut   Si A <> nul Alors
                Ecrire( A^.val ) ;
                Si A^.FG <> nul ou A^.FD <> nul Alors
```

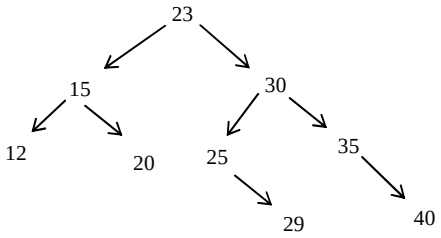
```
                    Ecrire( '( ') ;
                    affiche2 (A^.FG) ;
                    Ecrire(' , ') ;
                    affiche2 (A^.FD) ;
                    Ecrire(' )') ;
                Fsi ;
        Fsi ;
Fin ;
```

II. La pousse tirée par les cheveux (7)

Nous travaillons sur une application de recherche présentant comme particularité que les derniers éléments introduits sont les éléments les plus demandés. Nous souhaitons donc insérer dans un arbre de recherche un élément non pas comme une feuille mais à la racine afin que la recherche des derniers éléments introduits soit la plus rapide possible.

Voici par exemple un arbre binaire dans lequel nous insérons la valeur 23.

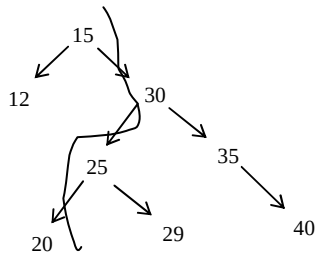




1) Vous devez écrire la séquence permettant de partager l'arbre existant en deux sous arbres contenant respectivement les éléments inférieurs et supérieurs à une valeur donnée.
Vous pouvez utiliser les opérations connues ainsi que l'opération **ajouter** de signature « ajouter(var tronc : arbre ; branche : arbre) » qui réalise un parcours de recherche et ajoute dans l'arbre désigné par tronc le sous arbre désigné par branche.
Le traitement ne doit pas entraîner la dégénérescence des sous arbres.

Nous réalisons un parcours de recherche qui permet de mettre à gauche tous les éléments inférieurs à la valeur référence et à droite les autres valeurs. Nous prenons soin de couper les liens sur le chemin de notre recherche. Certains liens seront rétablis comme 30-25.

```
tantque A <> nul faire
        Si A^.val < x  Alors
                ajouter(gauche, A) ;
                aux ← (A^.FD) ;
                A^.FD ← nul ;
        Sinon   ajouter(droite, A) ;
                aux ← (A^.FG) ;
                A^.FG ← nul ;
        Fsi ;
        A ← aux ;
Fait ;
```



Puisque nous parlions de séquence nous ne demandions pas de procédure récursive.
Voici toutefois quelques solutions récursives :

```
Procédure partage (A : ABR ; x : entier ; Var gauche, droite : ABR ) ;
Début
        Si A <> nul Alors
                Si A^.val < x  Alors
                        partage (A^.FD, x, gauche, droite) ;
                        A^.FD ← gauche ;
                        gauche ← A ;
                Sinon   partage (A^.FG, x, gauche, droite) ;
                        A^.FG ← droite ;
                        droite ← A ;
                Fsi ;
        Sinon   gauche ← nul ;
                droite ← nul ;
        Fait ;
Fin ;
```

Dans cette version on évite de couper les liens pour les rétablir par la suite. Toutefois le code est plus complexe et les tests plus nombreux lorsque l'on doit couper un lien.

```
Procédure partage2 (A : ABR ; x : entier ; Var gauche, droite : ABR ) ;
Début
        Si A <> nul Alors
                aux ← A ;
                Si A^.val < x  Alors
                        Tantque aux <> nul et aux^.val < x faire
                                père ← aux ;
                                aux ← aux^.fd ;
                        fait ;
```

```
                    gauche ← A ;
                    droite ← aux ;
                    Si aux <> null Alors
                        partager(aux^.FG, x, père^.FD, aux^.FG) ;
                    Fsi ;
                Sinon
                    Tantque aux <> nul et aux^.val > x faire
                        père ← aux ;
                        aux ← aux^.fg ;
                    fait ;
                    gauche ← aux ;
                    droite ← A ;
                    Si aux <> null Alors
                        partager(aux^.FD, x, aux^.FD, père^.FG) ;
                    Fsi ;
                Fsi ;
        Sinon   gauche ← nul ;
                droite ← nul ;
        Fsi ;
Fin ;
```

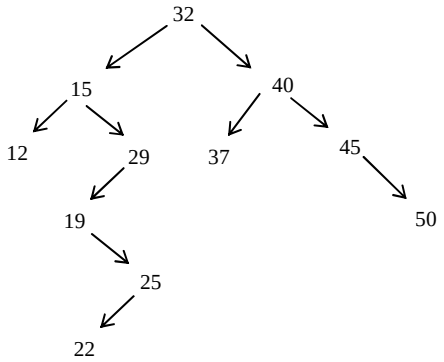
Une dernière version consiste à insérer la valeur dans l'arbre en tant que feuille et de remonter le nœud à chaque niveau par rotation. Toutefois cette version ne réalise par tout à fait la 1ère question puisque l'on traite en même temps la question 2.

```
Procédure insertion (Var A : ABR ; x : entier ) ;
Début
        Si A <> nul Alors
            Si A^.val < x  Alors
                Insertion (A^.FD, x ) ;
                RotationG ( A ) ;
            Sinon   Insertion (A^.FG, x ) ;
                    RotationD ( A ) ;

            Fsi ;
        Sinon   Nouveau(A) ;
                A^.val ← x ;
                A^.FG ← nul ;
                A^.FD ← nul ;
        Fait ;
Fin ;
```

2) Complètez votre séquence pour obtenir une procédure qui permet l'insertion d'une valeur dans l'arbre au niveau de la racine.

```
Procédure insertion ( Var A : ABR ; x : entier ) ;
Var     aux, gauche, droite : ABR ;
Début
        tantque A <> nul faire
            Si A^.val < x  Alors
                ajouter(gauche, a) ;
                aux ← (A^.FD) ;
                A^.FD ← nul ;
            Sinon   ajouter(droite, a) ;
                    aux ← (A^.FG) ;
                    A^.FG ← nul ;
            Fsi ;
            A ← aux ;
        Fait ;
        Nouveau(A)
        A^.val ← x ;
        A^.FD ← droite ;
        A^.FG ← gauche ;
Fin ;
```

3) Après avoir étudié les actions réalisées lorsque vous créez un nouvel arbre en insérant la valeur 23 dans l'arbre ci-dessous, modifiez votre procédure afin de réduire au maximum les parcours d'arbres.



L'analyse de l'exemple nous montre que chaque ajout dans un des deux arbres se fait à l'extrême droite pour le sous arbre gauche et à gauche pour le sous arbre droit. Ces nœuds sont les nœuds dernièrement ajouter à chaque arborescence.

```
Procédure insertion ( Var A : ABR ; x : entier ) ;
Var       gauche, droite, ga, da : ABR ;
Début
          gauche := nul ;
          droite := nul ;
          tantque A <> nul faire
                    Si A^.val < x  Alors
                              Si gauche = nul Alors gauche ← A ;
                              Sinon      ga^.fd ← A ;
                              Fsi
                              ga ← A ;
                              A ← A^.fd ;
                    Sinon     Si droite = nul Alors droite ← A ;
                              Sinon      da^.fg ← A ;
                              Fsi
                              da ← A ;
                              A ← A^.fg ;
                    Fsi ;
          Fait ;
          ga^.fd ← nul ;
          da^.fg ← nul ;
          Nouveau(A)
          A^.val ← x ;
          A^.FD ← droite ;
          A^.FG ← gauche ;
Fin ;
```

III. Le devoir de mémoire (courte) ! (5)

Quelles sont les deux composantes de la preuve de programmes et quels sont leurs rôles respectifs.

Quelles sont les caractéristiques d'un arbre 2-3-4 ?

Quelles sont les caractéristiques d'un arbre bicolore ou arbre rouge-noir ?

Donnez une application des arbres bicolores.