Chapitre 4 : Arbres Binaires

Un arbre est un ensemble de places, dites nœuds qui est organisé de façon hiérarchique à partir de son nœud initial, la racine (s'il n'est pas vide).

Les programmes informatiques sont transformés en arbres pour permettre leur traitement par le compilateur ou l'interpréteur : on parle par exemple d'arbre de syntaxe abstraite.

Ce sont encore des arbres qui permettent de décrire l'organisation des fichiers de systèmes d'exploitation comme Unix ou Linux, et des arbres toujours qui permettent d'optimiser les communications dans les réseaux, etc…

L'arbre est un outil algorithmique et plus généralement informatique essentiel.

Il existe de nombreuses sortes d'arbres et nous ne nous intéresserons qu'aux arbres binaires décorés, c'est-à-dire ceux pour lesquels les nœuds n'ont jamais plus de deux successeurs (fils) et contiennent une valeur.

1. Définition des arbres

Un arbre binaire décoré est :
- soit vide (noté ø ou vide ou Nil ou Null),
- soit de la forme <racine,arbre, arbre> où racine est une place (on dit aussi un nœud), arbre et arbre sont des arbres

Ce qui donne la forme algorithmique suivante :

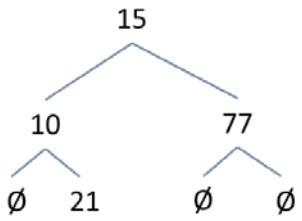
```
Arbre = ø
Racine = …
Arbre = <Racine, Arbre, Arbre>
```

Cette structure d'arbre est récursive, dès sa définition, mais aussi dans son exploitation.

• Racine, arbre gauche, arbre droit, fils, père, noeud, feuille

Soit un arbre non vide.
- racine est dit être sa racine
- les sous arbres sont dits être respectivement son sous-arbre gauche pour le premier et son sous-arbre droit pour le second.
- Si le sous-arbre gauche est non vide, sa propre racine est dite être un fils gauche de la racine initiale ; une définition analogue tient pour le sous-arbre droit et le fils droit.
- Ces deux fils sont dits être "frères ayant pour père la racine initiale".
- Ces définitions se transmettent tout au long de l'arbre.
- Les racines des sous-arbres sont appelées des nœuds.
- Une feuille est un nœud dont les fils sont vides, nommée encore nœud externe.
- Un nœud avec un ou deux fils non vides est dit nœud interne.

Exemple :



L'arbre A peut s'écrire = < 15 , < 10, ø, <21, ø, ø> > , <77, ø, ø> >
A a une racine dont le contenu est 15,
A a deux sous arbres g(A) = < 10, ø, <21, ø, ø> > à gauche et d(A) = <77, ø, ø>.
À droite, d(A) a une racine de valeur 77 et deux sous arbres vides.
À gauche, la racine de g(A) a pour valeur 10, le sous-arbre gauche de g(A) est vide, son sous-arbre droit est d(g(A)) = <21, ø, ø>, lui-même de racine 21, avec deux sous arbres vides.
Le nœud contenant 15 est père des nœuds contenant 10 et 77.
Le nœud contenant 21 est fils du nœud contenant 10.

Définitions

- Taille de l'arbre : nombre de ses nœuds, internes ou externes (feuilles) de l'arbre,
- Niveau (ou profondeur) d'un nœud : longueur du chemin qui le relie à la racine de l'arbre,
- Hauteur de l'arbre : le plus grand des niveaux de ses nœuds.

Sur notre exemple, la taille de A est égale à 4, le niveau du nœud contenant 77 est 1, la hauteur de A est 2.

Nous venons de donner des définitions concernant des arbres binaires, mais on utilise également couramment des arbres dont les nœuds ont un nombre variable de fils et éventuellement aucun contenu.

Lorsque les nœuds de l'arbre ont un contenu, c'est-à-dire que les nœuds sont des places contenant des éléments, on dit que l'arbre est étiqueté (ou décoré). Dans la suite, sauf mention contraire, nous parlerons d'arbres décorés, même si ce n'est pas explicitement mentionné. Il pourra arriver qu'un nœud soit désigné par abus de langage par son contenu (valeur).

2. Définition abstraite du type arbre

Type abstraits : Arbre utilise : Noeuds, Elément

Fonctionnalités de base, données d'emblée :

vide : on peut savoir si l'arbre est vide
vide?(A) ou A = Vide
racine : on sait repérer la place de la racine et son contenu
contenu : on peut connaître l'élément contenu dans le nœud (A.val en représentation chaînée)
gauche : sous-arbre gauche de l'arbre, c'est un arbre (A.g)
droite : sous-arbre droit de l'arbre, c'est un arbre (A.d)
faire_arbre : on sait construire un arbre à partir d'un élément et de deux arbres

La sémantique des fonctions

- L'arbre vide est un arbre qui n'a ni racine ni autres nœuds. On peut tester si un arbre est vide par la fonction vide? ou en utilisant le test A = vide ou A = Nil en représentation chaînée.
- Pour les représentations graphiques précédentes, la racine est la place du haut, qui contient une valeur quand l'arbre est étiqueté. La fonction contenu fournit cette valeur. Ici le nœud joue le rôle de la place chez les listes. Cependant, pour des raisons de facilité d'expressions, nous confondrons souvent la place et son contenu.
- Les fonctions gauche et droite (que nous noterons quelquefois g et d, ou arbre-gauche, arbre-droit) fournissent les sous arbres gauche et droit. Ceux-ci peuvent être vides, mais les fonctions ne sont définies que si l'arbre d'origine est non vide.
- La fonction faire_arbre a trois arguments : à partir d'un élément e et de deux arbres B et C, la fonction crée un arbre dont la racine contient e et dont le sous-arbre gauche est B et le sous-arbre droit est C.

- **Fonctionnalités de base des arbres**

Arbre réduit à une feuille

```
Fonction feuille ? (A : Arbre): Booléen
                                {rend vrai si l'arbre est réduit à une feuille, faux sinon}
Rendre non vide?(A) et vide?(gauche(A)) et vide?(droite(A))
```

NB : dans cette fonction, on suppose que l'évaluation de la condition est paresseuse. Vous pouvez, à titre d'exercice, rédiger la version non paresseuse.

Remplacer une valeur à la racine

```
Procédure remplacer_racine (x: élément ; var A : Arbre)
                                                    {si A est vide, ne fait rien}
                                {sinon remplace dans A la valeur de la racine par x}
Si non vide?(A) alors
        contenu(racine(A)) ← x
```

NB : l'arbre A est passé par adresse.

Variante :

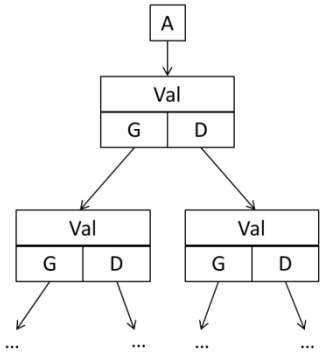
```
Procédure remplacer_bis_racine (x : élément ; var A : Arbre)
                                {si A est vide, donne un arbre de contenu racine x}
                                {si A est non vide, remplace le contenu de la racine par x}
Si (non vide?(A)) alors
        contenu(racine(A)) ← x
Sinon
        New (A)
        contenu(racine(A)) ← x
        gauche(A) ← Nil
        droite(A) ← Nil
```

Remplacer un sous-arbre

```
Procédure remplacer_à_gauche (G : Arbre; var A : Arbre)
                                        {si A est vide, ne fait rien}
                                    {sinon remplace dans A le sous-arbre gauche par G}
Si (non vide?(A)) alors
        gauche(A) ← G
```

3. Représentation chaînée des arbres

La représentation chaînée est la plus naturelle pour les arbres car elle permet une transcription directe des notions de nœud et de fils. L'arbre sera représenté par un pointeur, et à l'adresse repérée par le pointeur, il y aura un groupement contenant la valeur de la racine et deux autres pointeurs vers les fils, etc…



Définition de l'arbre "façon java", la récursivité des types est directe

```
Arbre = classe{
        Champs
                val : élément
                gauche, droite : Arbre
        Méthodes
                Contenu_racine() : élément
                Vide?() : booléen
}
```

Pour créer un arbre, on fait A = New (Arbre). Comme décrite précédemment, cette instruction :

- Réserve de la place pour la variable associée à A.
- Donne comme valeur à A l'adresse de cette place.

La variable A contient donc une adresse tandis que A.val contient un élément de type celui des contenus de racine. Et A.gauche et A.droite contiendront les adresses qu'on leur donnera (ou Nil).

Définition de l'arbre "façon Pascal", la récursivité des types est croisée

```
Type Arbre = ↑Nœuds
Nœuds = enregistrement
                val : élément
                gauche, droite : Arbre
```

- **Primitives des arbres**

Créer un arbre vide, tester s'il est vide

L'affectation A ← Nil crée un arbre vide, mais il n'y a pas de place réservée en mémoire. Si l'on fait New (A), cela réserve une place pour la variable repérée par A. L'adresse de cette variable est mise dans A, la variable associée, A↑ existe, même si elle ne contient rien.

```
Fonction vide? (A: Arbre): Booléen
                    {rend vrai si l'arbre est vide, faux sinon}
Rendre A = Nil
```

Trouver la racine d'un arbre

Il s'agit de la place désignée symboliquement par A↑. L'arbre est une adresse et la racine est ce qu'il y a à cette adresse.

```
Fonction Place_Racine (A: Arbre): Noeuds
                              {rend le noeud racine de l'arbre}
                              {Fonction interdite si A = Nil}
Rendre A↑
```

Obtenir le contenu d'un nœud

Si l'arbre est A, le contenu du nœud racine A↑ est A↑.val. En général, le contenu d'un nœud N est noté N.val.

```
Fonction Contenu (N : Noeuds) : élément
                              {rend le contenu du noeud}
                              {Fonction interdite si A = Nil}
Rendre N.val
```

Dans des langages comme C ou java, comme il n'y a pas de racine considérée comme un nœud, on accèdera directement au contenu :

```
Fonction Contenu-Racine (A : Arbre) : élément
                    {rend le contenu de la racine}
                    {Fonction interdite si A = Nil}
Rendre A.val
```

Trouver les sous-arbres gauche et droit

```
Fonction gauche (A : Arbre) : Arbre
                    {Fonction interdite si A = Nil}
                    {rend le sous-arbre gauche de A}
Rendre A↑.g


Fonction droite (A : Arbre) : Arbre
                    {Fonction interdite si A = Nil}
                    {rend le sous-arbre droit de A}
Rendre A↑.d
```

NB : dans des langages comme C ou java, A↑.g devient A.g et A↑.d devient A.d.

Faire un arbre à partir de deux sous-arbres et d'un élément-racine

Si l'on veut faire un nouvel arbre à partir d'un élément et de deux arbres, il faut réserver une nouvelle place, New (A), puis mettre dans A↑.val lavaleur du nouvel élément. Il faut mettre dans A↑.g la valeur A', mettre dans A↑.d la valeur A'' (les deux sous-arbres passés en paramètres).

```
Procédure Faire_arbre (var A : Arbre ; x: élément; A', A'': Arbre)
          {Construit un arbre, de racine contenant x et de sous-arbres A' et A''}
    Réserver une nouvelle place pour A                    {New(A)}
    Contenu(racine(A)) ← x          {si A existe avant, il est perdu définitivement}
    Gauche(A) ← A'
    Droite(A) ← A''
```

Une fois ces fonctions transcrites en langage informatique, tous les algorithmes et programmes peuvent s'écrire directement avec elles et il est quasiment possible d'oublier l'implémentation.

4. Acrobranche…

Les divers calculs à faire à propos des arbres nécessitent généralement de visiter les nœuds, c'est-à-dire de les parcourir.
Nous utiliserons un parcours classique, le parcours en profondeur d'abord, à main gauche.
Il consiste à considérer l'arbre comme une main fort ramifiée, et à tourner autour au plus près à partir de la gauche. On démarre de la racine et l'on parcourt successivement le sous-arbre gauche, puis le sous-arbre droit, récursivement.
Les fonctions utilisant ce type de parcours feront à chaque nœud, un traitement quelconque (par exemple compter, ou écrire, ou vérifier une propriété quand il s'agit de model-checking…) et deux appels récursifs appliqués aux sous-arbres gauche et droit.

- **Fonctions de base**

Calculer la taille d'un arbre

La taille d'un arbre est le nombre de ses nœuds (internes et externes).
Si l'arbre est vide, sa taille est zéro ; sinon, c'est un nœud de plus que la somme du nombre de nœuds des sous-arbres gauche et droit.
La taille est donnée par la formule récursive :

taille (vide) = 0
taille (<racine, arbre_g, arbre_d>) = 1 + taille(arbre_g) + taille(arbre_d)

```
Fonction taille (A : Arbre) : entier
                                {rend le nombre de nœuds de l'arbre}
si vide?(A) alors
        rendre 0
sinon rendre 1 + taille(gauche(A)) + taille(droite(A))
```

Complexité : de l'ordre de la taille de l'arbre (pour chaque nœud, on a une comparaison et deux additions)

Faire la somme des valeurs des nœuds

Si l'arbre est vide, alors la somme est nulle, sinon elle vaut la somme de la valeur du nœud et des sommes des valeurs dans l'arbre gauche et l'arbre droit.

Formule récursive :

somme (vide) = 0
somme (<racine, arbre_g, arbre_d>) = valeur(contenu(racine)) + somme(arbre_g) + somme(arbre_d)

Algorithme en version "représentation chaînée" :

```
Fonction somme_noeuds (A : Arbre) : élément
                                {on suppose savoir faire l'addition d'éléments}
si A = nil alors
        rendre 0
sinon rendre A.val + Somme_noeuds (A.g) + Somme_noeuds (A.d)
```

La complexité en nombre de visites est de l'ordre du nombre de nœuds, *ie* la taille de l'arbre (donc linéaire).

Recherche d'un élément dans un arbre

Il s'agit de savoir si un élément donné est identique la valeur d'un des nœuds internes ou externes de l'arbre, et si oui, duquel.
Nous présentons ici un algorithme naïf, en attendant d'identifier des structures d'arbres plus adaptés à la recherche.
Il s'agit de parcourir l'arbre en commençant par la racine : si élém est la valeur de la racine, on renvoie vrai. Sinon, on recherche élém dans le sous-arbre gauche. Si l'on n'a pas trouvé la valeur dans le sous-arbre gauche, on la recherche ensuite dans le sous-arbre droit.

```
Fonction cherche (élém : élément ; A : Arbre) : Booléen
                        {rend vrai si l'élément est contenu d'un nœud de l'arbre}
si vide?(A) alors
        rendre faux
sinon
        si (contenu(racine(A)) = élém) alors rendre vrai
        sinon
                si cherche(élém, gauche(A)) alors rendre vrai
                sinon rendre cherche(élém, droite(A))
```

Si l'on souhaite obtenir l'adresse de l'élément dans l'arbre, il faut modifier la fonction :

```
Fonction cherche_at(élém : élément ; A : Arbre) : Arbre
                        {rend Nil ou un nœud dont le contenu est l'élément}
                        /* utilise une variable locale P1 : Arbre* /
si vide?(A) alors
        rendre Nil
sinon
        si (contenu(racine(A)) = élém) alors rendre racine(A)
        sinon   P1 ← cherche_at(élém, gauche(A))
                si (non vide?(P1)) alors rendre P1
                sinon rendre cherche_at(élém, droite(A))
```

Complexité en nombre de visites égale à la taille de l'arbre.

Hauteur de l'arbre

La hauteur d'un arbre est la plus grande longueur des chemins allant de la racine aux feuilles.
Si l'arbre est réduit à la racine, c'est zéro ; sinon, c'est 1 de plus que la plus grande des deux hauteurs des sous-arbres gauche et droit.
Par convention et parce que cela rend les calculs cohérents, l'arbre vide a pour hauteur -1.

Formule récursive :

hauteur (vide) = -1
hauteur (<rac, arbre_g, arbre_d>) = 1 + Maximum(hauteur (arbre_g), hauteur(arbre_d))

```
Fonction hauteur (A : Arbre) : entier
                                        {retourne la hauteur de l'arbre}
                                        {utilise la fonction auxiliaire Maxi}
        si vide?(A) alors rendre -1
        sinon rendre 1 + Maxi(hauteur(gauche(A)), hauteur(droite(A)))


Fonction Maxi (N, M : entier) : entier
                                {retourne le maximum des deux nombres}
        si N > M alors rendre N
        sinon rendre M
```

La complexité en nombre de visites est égale à la taille de l'arbre (linéaire).

- **Les divers parcours des arbres binaires…**

Nous allons voir ici deux des nombreuses façons de parcourir un arbre binaire : le parcours en profondeur et le parcours en largeur.

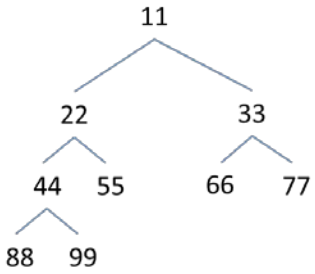
Parcours en profondeur

Le parcours en profondeur à main gauche est un parcours classique.
On tourne autour de l'arbre à partir de la gauche : on est à la racine, puis on se promène successivement dans le sous-arbre gauche, puis dans le sous-arbre droit.
On rencontre ainsi chaque nœud de l'arbre trois fois :

- une fois en descendant, on est à sa gauche,
- puis en position médiane, on est en dessous de lui,
- puis en remontant, on est à droite

Il y a donc en général deux appels récursifs, mais l'ordre dans lequel on travaille sur les nœuds dépend de la place que l'on donne à ces appels récursifs relativement aux autres instructions qui sont à faire sur le nœud racine du sous-arbre visité.

Exemple :



Pour mémoire, le niveau d'un nœud est la longueur du chemin qui le relie à la racine.
L'arbre ci-dessus a :

- un nœud au niveau 0 (valeur : 11)
- deux nœuds au niveau 1 (valeurs : 22 et 33)
- quatre nœuds au niveau 2 (valeurs : 44, 55, 66, 77)
- deux nœuds au niveau 3 (valeurs : 88 et 99)

Voici l'ordre dans lequel on affiche ces nœuds lors d'un parcours en profondeur à main gauche, selon la position de la commande « écrire ».

*1° à la descente : on traite le nœud avant de descendre dans le sous-arbre gauche. On parle d'ordre **préfixe**.*

```
Procédure Écrire_Parcours_Préfixe (A : Arbre)
Si non (vide?(A)) alors
        écrire Contenu(Racine(A))
        Écrire_Parcours_Préfixe (gauche(A))
        Écrire_Parcours_Préfixe (droite(A))
```

Résultat sur l'arbre-exemple : 11, 22, 44, 88, 99, 55, 33, 66, 77.

*2° en position médiane : on traite le nœud après avoir parcouru son fils gauche et avant d'avoir parcouru son fils droit. On parle d'ordre **infixe** (ou symétrique).*

```
Procédure Écrire_Parcours_Infixe (A : Arbre)
Si non (vide?(A)) alors
        Écrire_Parcours_Infixe (gauche(A))
        écrire Contenu(Racine(A))
        Écrire_Parcours_Infixe (droite(A))
```

Résultat sur l'arbre-exemple : 88, 44, 99, 22, 55, 11, 66, 33, 77.

*3° en remontant : on traite le nœud après avoir parcouru ses fils gauche et droit. On parle d'ordre **suffixe** ou **postfixe**.*

```
Procédure Écrire_Parcours_Postfixe (A : Arbre)
Si non (vide?(A)) alors
        Écrire_Parcours_Postfixe (gauche(A))
        Écrire_Parcours_Postfixe (droite(A))
        écrire Contenu(Racine(A))
```

Résultat sur l'arbre-exemple : 88, 99, 44, 55, 22, 66, 77, 33, 11.

Remarque : évidemment, la plupart du temps, on réalisera de "vrais" traitements sur la valeur portée par le nœud au lieu de se contenter de l'écrire.

<u>Parcours Préfixe itératif utilisant une pile</u>

On peut implémenter le parcours préfixe en utilisant une pile : on part de la racine, que l'on traite (ici, c'est une écriture), puis on empile la racine du sous-arbre droit, suivie de celle du sous-arbre gauche. On dépile (l'élément du haut), on l'écrit, puis on empile la racine de son sous-arbre droit, suivie de celle de son sous-arbre gauche. On dépile, on écrit, etc… on va ainsi écrire, empiler et dépiler tous les nœuds du sous-arbre gauche avant d'attaquer tous ceux du sous-arbre droit. Le résultat est une écriture préfixe des nœuds de l'arbre.

Voici un algorithme possible :

```
Procédure Écrire_Préfixe (A : Arbre)
                        {Écrit les valeurs des nœuds dans l'ordre préfixe}
Si A = vide alors
        {ne rien faire}
Sinon
        Créer une pile P et y empiler la racine de A
Tant que P ≠ Vide faire
        noeud_courant ← sommet (P)
        Afficher {traiter} la valeur du contenu de noeud_courant
        Dépiler (P)
        Si le sous-arbre droit de noeud_courant est non vide, alors
                l'empiler dans P
        Si le sous-arbre gauche de noeud_courant est non vide, alors
                l'empiler dans P
```

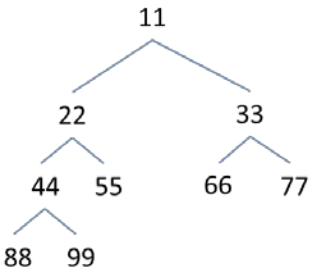
Exercice : rédiger chacune des étapes pour l'arbre-exemple ci-dessus.

On voit bien ici que la pile permet de mémoriser un travail à accomplir plus tard : on empile les sous-arbres droits qui ne seront dépilés que longtemps après leurs descendants, et en ordre inverse, tandis que les sous-arbres gauches sont dépilés dans l'ordre de descente.

On peut évidemment rédiger le même algorithme en représentation chaînée : c'est un très bon exercice à faire !

Parcours en largeur

Il s'agit ici de parcourir l'arbre par niveaux, et donc dans sa largeur : les nœuds sont lus de gauche à droite.
Pour l'arbre-exemple, on rencontre, dans l'ordre : 11, 22, 33, 44, 55, 66, 77, 88, 99.



Pour ce type de parcours, on peut facilement utiliser une file : on commence par le niveau 0, qui donne la racine, puis, à chaque étape k, on met dans la file la liste ordonnée gauche - droite des nœuds de niveau k, puis on les prélève en tête un à un ; pour chacun d'eux on écrit (traite) sa valeur et on met dans la file (en queue) le sous-arbre gauche, puis le sous-arbre droit : cela fournit la liste ordonnée des nœuds de niveau k et dans la file il y a la liste ordonnée des nœuds de niveau 1+k qui permet de continuer.
Le principe est simple : il suffit de stoker à chaque niveau la liste dans une file FIFO.

Un algorithme possible en représentation chaînée :

```
Type    Arbre = enregistrement
                  val : élément
                  g, d : Arbre

        Liste = enregistrement
                  noeud : Arbre
                  suiv : Liste

        File = enregistrement
                  Tête, queue : Liste

Procédure Mettre_la_racine (A : Arbre ; var F : File)
                                          {Créer une file et y placer la racine de l'arbre}
        Créer(L)                              {variable locale L, une nouvelle liste}
        L.noeud ← A                           {On y place l'arbre A}
        L.suiv ← Nil                          {Il n'a pas de suivant dans la liste}
        F.tête ← L                  {on dit que c'est la tête et la queue de la nouvelle file}
        F.queue ← L
{fin_procédure}

Fonction Vide? (F : File) : Booléen
        Si (F.tête = Nil) ou (F.queue = Nil) alors
                  retourner Vrai
        Sinon retourner faux
        {fin_fonction}
```

```
Procédure Insérer (X : Arbre ; var F : File)
                                                    {on insère X en queue de file}
        Créer(L)                                    {variable locale L qui sera mise à la fin}
        L.noeud ← X                                 {On y place l'arbre X à insérer}
        L.suiv ← Nil                                {Il n'a pas de suivant dans la liste}
        F.queue.suiv ← L                            {on l'attache à la fin de la file}
        F.queue ← L                                 {on déplace la queue d'un cran vers la fin}
{fin_procédure}

Procédure Décapiter (var F : File)
        Si non (Vide?(F)) alors F.tête ← F.tête.suiv      {on décale la tête d'un cran en arrière}
{fin_procédure}

Procédure Écrire_Largeur (A : Arbre)
                                                    {Écrit les valeurs des noeuds en largeur}
        Si A = vide alors {ne rien faire}
        Sinon
                Mettre_la_racine (A, F) {on met la racine dans une file F variable locale}
                Tant que non Vide ? (F) faire
                        noeud_courant ← F.tête.noeud {variable locale}
                        Écrire(noeud_courant.val)  {on écrit la valeur du noeud en tête de file}
                        Décapiter (F)              {on l'enlève de la tête de file, traitée}
                              {on recharge éventuellement la file avec les fils du noeud enlevé}
                        Si noeud_courant.g ≠ Nil alors Insérer (noeud_courant.g, F)
                        Si noeud_courant.d ≠ Nil alors Insérer(noeud_courant.d, F)
{fin_procédure}
```

Sur l'arbre-exemple ci-dessus, le déroulement se passe comme suit…

- On met 11 dans la file
- On prend 11 en tête de file ; on le traite ; on met dans la file : 22, puis 33.
- On prend 22 en tête de file ; on le traite ; on met dans la file les fils de 22 : 44, puis 55. Ici, la file contient (à partir de la droite, *i.e.* la tête) 33, 44, 55.
- On prend la tête, 33 ; on la traite ; on met dans la file ses fils : 66, puis 77. La file contient maintenant (de droite à gauche) 44, 55, 66, 77.
- On prend la tête, 44 ; on met dans la file ses fils : 88, puis 99. La file contient 55, 66, 77, 88, 99. Ils seront retirés de la file par la tête sans qu'on ne rajoute plus rien, donc dans l'ordre : 55, 66, 77, 88, 99. On a bien obtenu l'ordre cherché.