

Epita:Algo:Cours:Info-Sup:Structures arborescentes

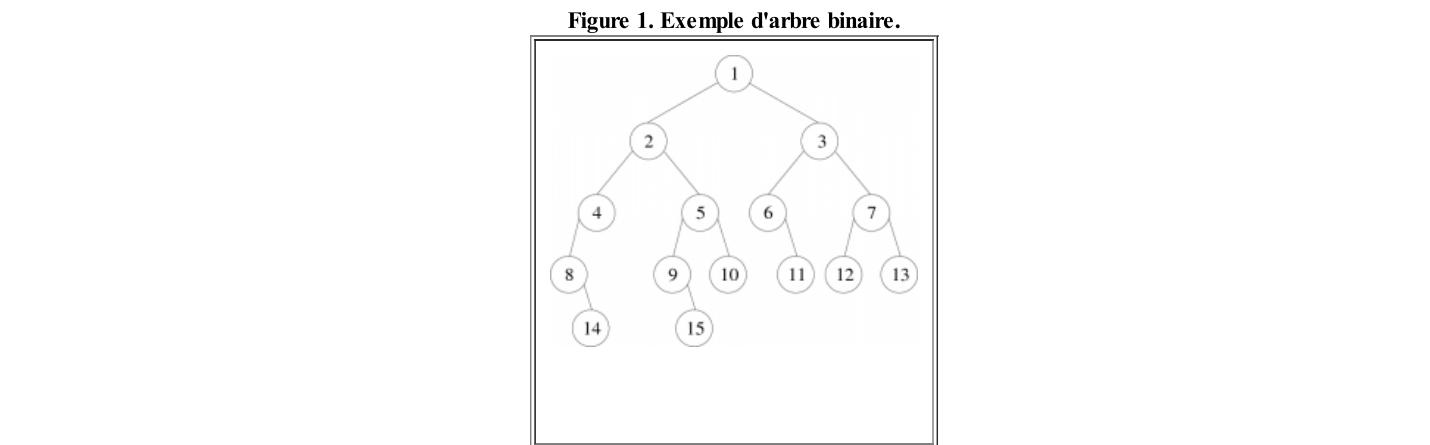
De EPITACoursAlgo.

Sommaire

- 1 Les arbres binaires
 - 1.1 Le type Arbre binaire
 - 1.1.1 Terminologie
 - 1.1.2 Mesures sur les arbres binaires
 - 1.1.3 Arbres binaires particuliers
 - 1.1.4 Occurrence et numérotation hiérarchique
 - 1.2 Représentation des arbres binaires
 - 1.2.1 Représentation dynamique
 - 1.2.2 Représentation statique
 - 1.3 Parcours d'un arbre binaire
 - 1.3.1 Parcours en profondeur
 - 1.3.1.1 Algorithme de parcours profondeur main gauche
 - 1.3.1.2 Parcours profondeur d'un arbre étiqueté représentant une expression arithmétique
 - 1.3.2 Parcours en largeur
- 2 Les arbres généraux
 - 2.1 Le type Arbre général
 - 2.2 Représentation des arbres généraux
 - 2.2.1 Représentation sous forme statique-dynamique
 - 2.2.2 Représentation sous forme dynamique
 - 2.2.3 Représentation sous forme d'arbre binaire
 - 2.2.4 Représentation sous forme de N-uplet
 - 2.3 Parcours d'un arbre général
 - 2.3.1 Parcours en profondeur
 - 2.3.2 Parcours en largeur

Les arbres binaires

Un arbre binaire est une structure par nature récursive. Il peut être soit vide (\emptyset), soit la composée d'un noeud racine et de deux sous-arbres ($< o, G, D >$) où o est le noeud racine, et G et D sont deux arbres binaires disjoints (respectivement sous-arbre gauche et sous-arbre droit).



Dans cet exemple, nous avons associé à chaque noeud un numéro pour pouvoir les distinguer les uns des autres. Une chose importante à remarquer est la non symétrie d'un arbre binaire.

Le type *Arbre binaire*

```

types
  arbrebinnaire
utilise

```

noeud, élément

opérations

```

arbrevide : → arbrebinaire
<_,_,_> : noeud × arbrebinaire × arbrebinaire → arbrebinaire
racine : arbrebinaire → noeud
g : arbrebinaire → arbrebinaire
d : arbrebinaire → arbrebinaire
contenu : noeud → élément

```

préconditions

```

racine(B) est-défini-ssi B ≠ arbrevide
g(B) est-défini-ssi B ≠ arbrevide
d(B) est-défini-ssi B ≠ arbrevide

```

axiomes

```

racine (<r,G,D>) = r
g (<r,G,D>) = G
d (<r,G,D>) = D

```

avec

```

noeud r
arbrebinaire B,G,D

```

On peut remarquer en bleu l'opération et le type nécessaire à un arbre étiqueté.

Terminologie

La manipulation des arbres binaires nécessite l'utilisation d'un vocabulaire approprié dont voici les principaux éléments :

Soit un arbre binaire $\mathbf{B} = \langle \mathbf{r}, \mathbf{G}, \mathbf{D} \rangle$

- \mathbf{r} est le noeud racine de \mathbf{B}
- \mathbf{G} est le sous-arbre gauche de \mathbf{B}
- \mathbf{D} est le sous-arbre droit de \mathbf{B}
- On appelle fils gauche (*fils droit*) d'un noeud \mathbf{ni} , la racine du sous-arbre gauche (*sous-arbre droit*) de l'arbre dont le noeud \mathbf{ni} est racine
- Le lien entre un noeud et son fils gauche (*fils droit*) est appelé lien gauche (*lien droit*)
- Un noeud \mathbf{ni} (seulement quand il approche de l'écurie) ayant pour fils gauche (*fils droit*) un noeud \mathbf{nj} est appelé père de \mathbf{nj}
- Deux noeuds de même père sont dits frères
- Un noeud \mathbf{ni} est appelé ascendant (*descendant*) d'un noeud \mathbf{nj} si, et seulement si, \mathbf{ni} est le père (*fils*) de \mathbf{nj} ou un ascendant (*descendant*) du père (*fils*) de \mathbf{nj}
- Les noeuds d'un arbre binaire ont au plus deux fils
- Un noeud ayant deux fils est appelé noeud interne ou point double
- Un noeud n'ayant qu'un fils gauche (*fils droit*) est appelé point simple à gauche (*point simple à droite*), ou noeud interne
- Un noeud n'ayant pas de fils est appelé noeud externe ou feuille
- Tout chemin allant de la racine de \mathbf{B} à une feuille de \mathbf{B} est appelé branche de \mathbf{B}
- Un arbre binaire possède autant de branches que de feuilles
- Le chemin obtenu en partant de la racine et ne suivant que des liens gauches (*liens droits*) est appelé bord gauche de \mathbf{B} (*bord droit de B*)

Remarques diverses : Contrairement à la vraie vie, les arbres binaires ne présentent pas de variétés à feuillage persistant ou caducue. Quand je veux supprimer une feuille, je le fais. Mais le plus fort, c'est que j'en crée une (de feuille) quand je veux. On oubliera aussi les notions familiales de cousinage et autres.

Mesures sur les arbres binaires

Nous allons voir maintenant des opérations sur les arbres qui vont nous permettre de prendre diverses mesures et de pouvoir alors déterminer la complexité des différents algorithmes qui leur sont appliqués (aux arbres évidemment, de quoi on parle là ?).

Ces opérations sont les suivantes :

- La taille d'un arbre \mathbf{B} correspond au nombre de ses noeuds, elle est définie par :

$$T(\mathbf{B}) = \begin{cases} 0 & \text{si } \mathbf{B} \text{ est un arbre vide} \\ 1 + g(\mathbf{B}) + d(\mathbf{B}) & \text{sinon} \end{cases}$$

- La hauteur, profondeur ou niveau d'un noeud x d'un arbre \mathbf{B} est définie par :

$$H(x) = \begin{cases} 0 & \text{si } x \text{ est la racine de } \mathbf{B} \\ H(père(x)) + 1 & \text{sinon} \end{cases}$$

$$\left\{ \begin{array}{l} 1+H(y) \text{ si } y \text{ est le père de } x \end{array} \right.$$

- La hauteur, profondeur d'un arbre **B** est définie par :

$H(B)=\max(H(x))$ avec **x** les noeuds de **B**

- La longueur de cheminement d'un arbre **B** est définie par :

$LC(B)=\sum H(x)$ avec **x** les noeuds de **B**

- La longueur de cheminement externe d'un arbre **B** est définie par :

$LCE(B)=\sum H(xe)$ avec **xe** les noeuds externes (*feuilles*) de **B**

- La longueur de cheminement interne d'un arbre **B** est définie par :

$LCI(B)=\sum H(xi)$ avec **xi** les noeuds internes (*points simples ou doubles*) de **B**

On a alors la relation : **LC(B)=LCE(B)+LCI(B)**

- La profondeur moyenne d'un arbre **B** est définie par :

$PM(B)=LC(B)/T(B)$

- La profondeur moyenne externe d'un arbre **B** est définie par :

$PME(B)=LCE(B)/Nbe$ avec **Nbe** le nombre de noeuds externes de **B**

- La profondeur moyenne interne d'un arbre **B** est définie par :

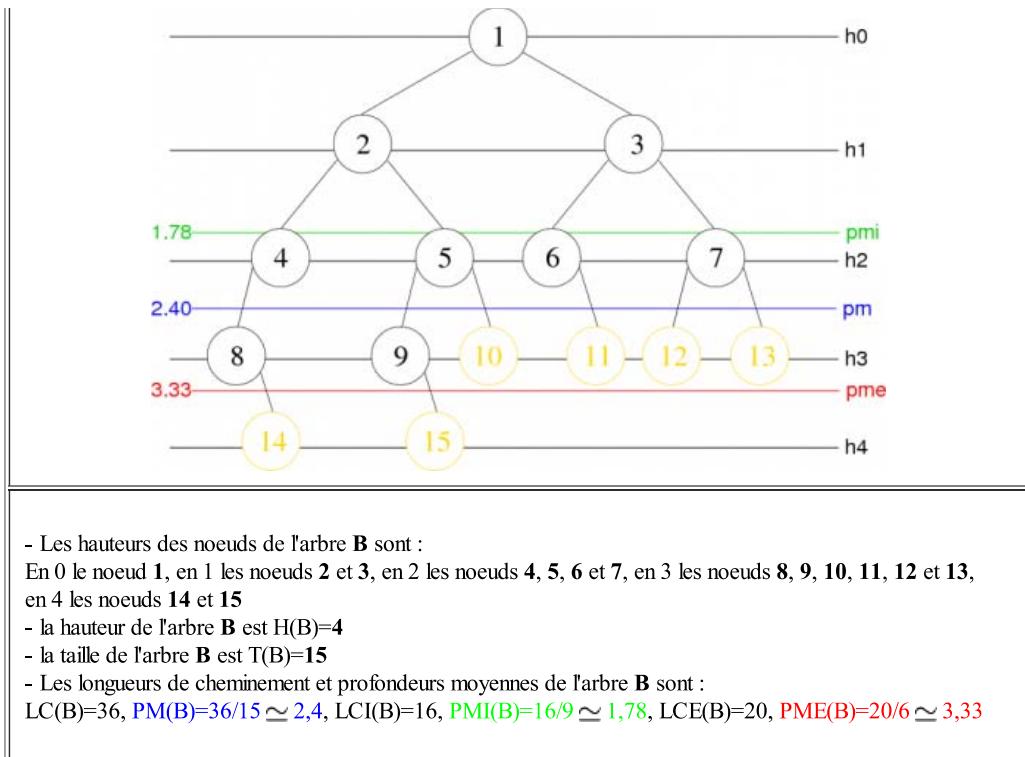
$PMI(B)=LCI(B)/Nbi$ avec **Nbi** le nombre de noeuds internes de **B**

Note : Des mesures comme la longueur de cheminement et la profondeur moyenne seront très utile pour déterminer la complexité des algorithmes appliqués aux arbres binaires.

Pour clarifier tout cela, nous allons prendre l'arbre de la figure 1 et préciser ses caractéristiques et mesures. Pour cela nous l'appellerons cet arbre **B** et conserverons la numérotation des noeuds de l'arbre, ce qui donne :

Figure 2. Exemple de mesures prises sur l'arbre de la figure 1.

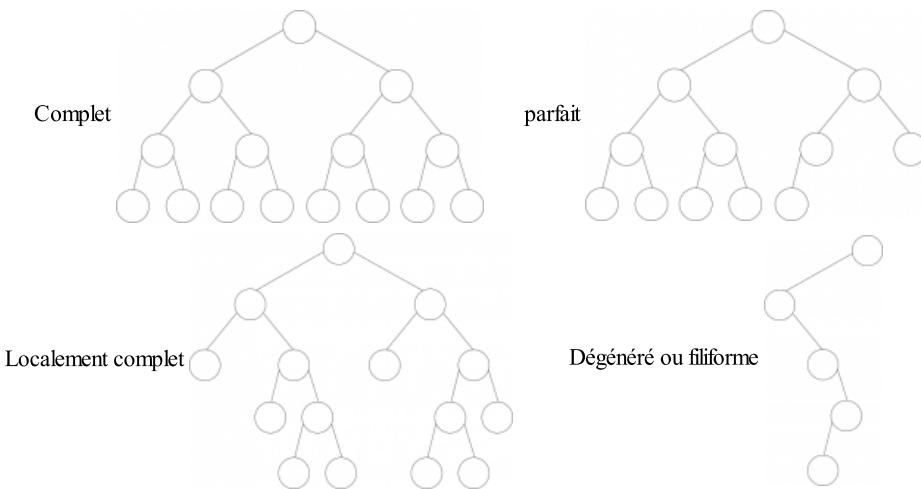
<ul style="list-style-type: none"> - 1 est la racine de B, 2 est son fils gauche et 3 son fils droit - les frères sont : (2,3), (4,5), (6,7), (9,10), (12,13) - 1, 2, 3, 5 et 7 sont des points doubles de B - 4 est un point simple à gauche de B - 6, 8 et 9 est un point simple à droite de B - 10, 11, 12, 13 et 14 et 15 sont des feuilles de B - (1, 2, 4, 8) et (1, 3, 7, 13) sont les bords gauches et droits de B - (1, 2, 4, 8, 14), 1, 2, 5, 9, 15), (1, 2, 5, 10), (1, 3, 6, 11), (1, 3, 7, 12) et (1, 3, 7, 13) sont les branches de B



Arbres binaires particuliers

Il existe des formes particulières d'arbres binaires qu'il faut connaître dans la mesure où leur spécificité permet de modifier les algorithmes sur les arbres, voire d'utiliser des algorithmes propres à leur structure. Ces arbres présentés en figure 3 sont :

Figure 3. Arbres binaires particuliers.



- Les arbres dégénérés ne sont constitués que de points simples à gauche ou à droite.
- Les arbres complets voient tous leurs niveaux remplis
- Les arbres parfaits voient tous leurs niveaux remplis excepté le dernier qui est rempli de gauche à droite.
- Les arbres localement complets ne sont constitués que de points doubles et de feuilles. il existe des arbres localement complets particuliers comme le peigne droit (*peigne gauche*) dont tous les fils gauches (*droits*) sont des feuilles.

Occurrence et numérotation hiérarchique

Une façon de décrire un arbre binaire est de lui associer un mot formé de **0** et de **1**. Ce mot est appelé occurrence du noeud. Par définition, la racine d'un arbre est noté **E** et si un noeud a pour occurrence **m** alors, son fils gauche à pour occurrence **m0** et son fils droit **m1**.

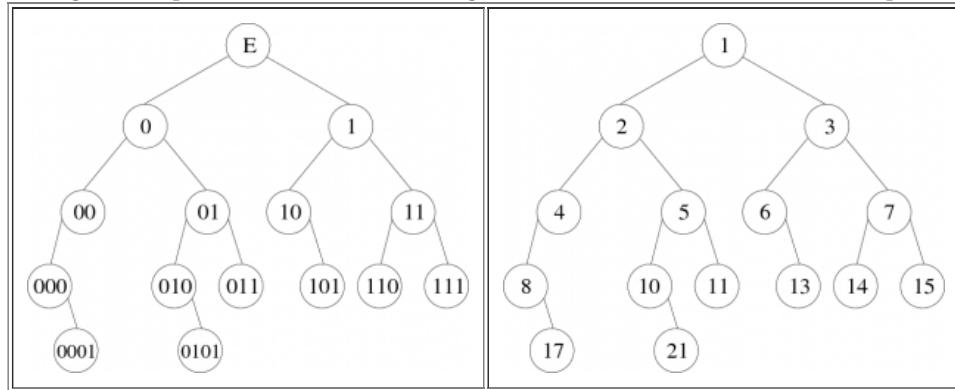
Pour l'arbre de la figure 1 appelé **B**, nous aurions alors :

$$B = \{E, 0, 1, 00, 01, 10, 11, 000, 010, 011, 101, 110, 111, 0001, 0101\}$$

D'autre part, les noeuds peuvent être numérotés de façon hiérarchique. C'est à dire que si l'on a un noeud **ni** alors son fils gauche sera le noeud **n2i** et son fils droit le noeud **n2i+1**, la racine ayant toujours le numéro **1**.

Ce qui pour l'exemple de la figure 1 donnerait :

Figure 4. Représentation de l'arbre de la figure 1 sous forme d'occurrence et hiérarchique.



Remarque : Nous verrons plus loin que la numérotation hiérarchique présente, entre autres, un intérêt sur certains arbres lors d'une représentation statique.

Représentation des arbres binaires

Comme pour les structures séquentielles, nous avons la possibilité de représenter les arbres en mémoire sous forme dynamique ou sous forme statique. Cette dernière pouvant être utilisée pour simuler la première.

Représentation dynamique

Cette représentation est quasiment un calque de la structure d'un arbre binaire. Chaque Noeud contient un élément (l'étiquette) et deux liens : un vers le fils gauche et l'autre sur le fils droit. Ce qui donne :

```

Types

t_element = ...
t_arbre = ↑t_noeud
t_noeud = enregistrement
    t_element elt
    t_arbre fg, fd
fin enregistrement t_noeud

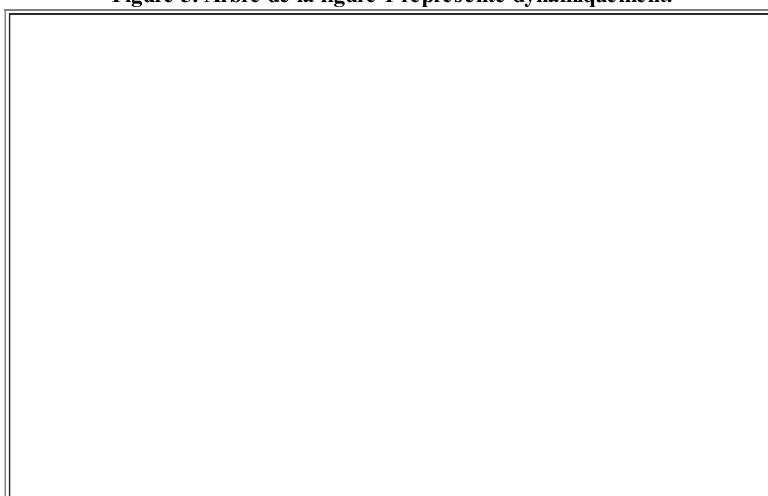
Variables

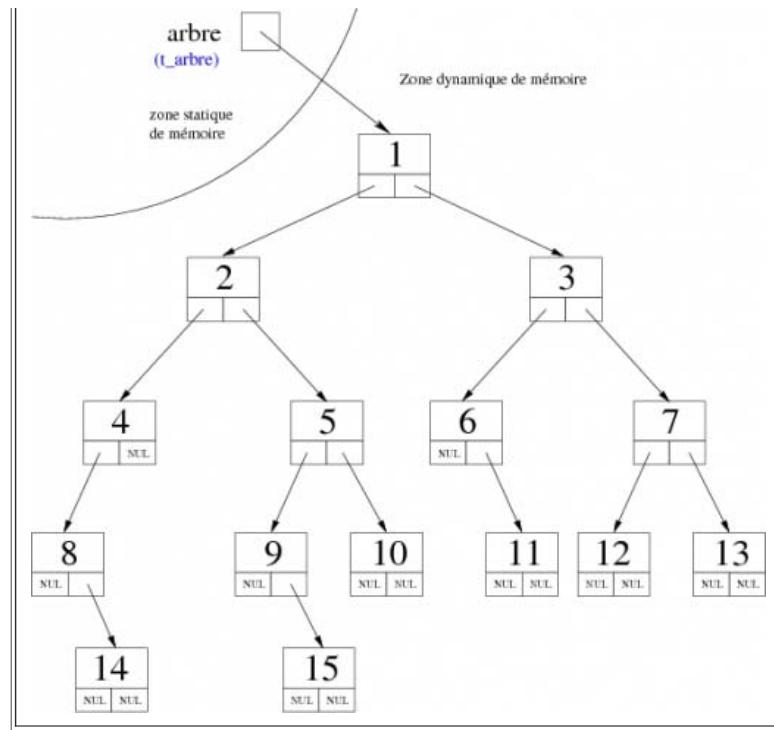
t_arbre arbre

```

Ce qui correspondrait à la structure suivante :

Figure 5. Arbre de la figure 1 représenté dynamiquement.





Dans ce type de représentation, les correspondances entre les opérations abstraites et l'implémentation dynamique du type (pour un arbre **B**) sont les suivantes :

Type abstrait \leftrightarrow Implémentation dynamique

$B = \text{Arbre-vide}$	$B = \text{nul}$
$B \leftarrow \langle _, _, _ \rangle$	Allouer(B)
racine(B)	$B \uparrow$
g(B)	$B \uparrow.\text{fg}$
d(B)	$B \uparrow.\text{fd}$
Contenu(racine(B))	$B \uparrow.\text{Elt}$

Représentation statique

Dans cette représentation, l'arbre sera contenu dans un vecteur (tableau à une dimension) d'enregistrements ayant chacun 3 champs : l'élément et deux entiers (un pour le fils gauche et l'autre pour le fils droit). Ces derniers (les deux entiers) référencant dans le vecteur l'indice du fils concerné (Suis-je assez clair ?).

La base du Vecteur étant 1, nous pouvons utiliser 0 comme référence de pointeur nul (l'arbre vide, quoi ?!). De plus, il nous faut la position de la racine donnée par un entier.

Remarque : Lors d'une implémentation en C, il faudra alors utiliser $-I$, la base étant 0.

Cela étant dit, voici la déclaration algorithmique d'une telle structure :

```

Constantes
Nbmax = 21

Types
t_element = ...
t_noeud = enregistrement
  t_element elt
  entier fg, fd
fin enregistrement t_noeud

t_vectNbmaxnoeud = Nbmax t_noeud
t_arbre = enregistrement
  t_vectNbmaxnoeud noeuds
  entier racine
fin enregistrement t_arbre

Variables

```

t_arbre arbre

Dans ce type de représentation, les correspondances entre les opérations abstraites et l'implémentation statique du type (pour un arbre **B**) sont les suivantes :

Type abstrait	\Leftrightarrow	Implémentation statique
$B = \text{Arbre-vide}$		$B.\text{racine} = 0$
$\text{racine}(B)$		$B.\text{racine}$
$g(B)$		$B.\text{noeuds}[B.\text{racine}].\text{fg}$
$d(B)$		$B.\text{noeuds}[B.\text{racine}].\text{fd}$
$\text{Contenu}(\text{racine}(B))$		$B.\text{noeuds}[B.\text{racine}].\text{elt}$

Adaptée à l'arbre de la figure 1, nous aurions le tableau présenté en **figure 6(a)**. En fait, dans ce type d'arbre, la racine peut être située à n'importe quel indice. Ce qui fait que l'on peut représenter dans un même tableau plusieurs arbres en même temps, une forêt par exemple (Et là, je suis très sérieux.). Il suffit simplement de maîtriser pour chacun la position de sa racine.

Figure 6. Représentation statique d'un arbre binaire

	elt	fg	fd
1	1	2	3
2	2	4	5
3	3	6	7
4	4	8	0
5	5	9	10
6	6	0	1
7	7	12	13
8	8	0	14
9	9	0	15
10	10	0	0
11	11	0	0
12	12	0	0
13	13	0	0
14	14	0	0
15	15	0	0
16			
17			
18			
19			
20			
21			

	elt
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	
10	9
11	0
12	0
13	0
14	12
15	13
16	
17	14
18	
19	
20	
21	15

	elt
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	
10	9
11	10
12	
13	11
14	12
15	13
16	
17	14
18	
19	
20	
21	15

(a)

(b)

(c)

Maintenant, si nous avions utilisé, pour le même arbre, une numérotation hiérarchique des noeuds (pas des étiquettes), nous aurions obtenu le tableau de la **figure 6(b)**.

Cette numérotation présente deux défauts majeurs :

- Elle nous oblige à fixer la racine d'un arbre non vide en indice 1
- Elle génère des trous dans le tableau et par conséquent un taux d'occupation assez faible.

Cela dit, elle est idéale pour les arbres complets ou parfaits dans la mesure où, pour ceux-ci tous les niveaux, sauf éventuellement le dernier, sont remplis, donc pas de trous (Golfeurs du Monde, Désolé !).

De plus, un noeud **ni** ayant pour fils gauche un noeud **n2i** et pour fils droit un noeud **n2i+1**, nous n'avons plus besoin de référencer le fils gauche et le fils droit. Ce qui permet de représenter l'arbre à l'aide d'un simple vecteur d'éléments comme montré sur la **figure 6(c)**.

Remarques :

- Il faut trouver un moyen à l'aide de l'élément pour référencer un arbre vide (la racine à 0, par exemple).
- Cette représentation permet, sans avoir à le mémoriser, de connaître le père d'un noeud. Il suffit pour un noeud **ni** avec $i > 1$ de rechercher le noeud **ni div 2**.
- Nous sommes toujours tenus d'avoir une racine en 1 pour un arbre non vide.

Pour cette représentation (**figure 6(c)**), la déclaration algorithmique deviendrait :

```

Constantes
Nbmax = 21

Types
t_element = ... /* Définition du type des éléments */
t_vectNbmaxnoeud = Nbmax t_element /* Définition du tableau des noeuds */
t_arbre = enregistrement /* Définition du type t_arbre */
  t_vectNbmaxnoeud noeuds
  entier      racine
fin enregistrement t_arbre

Variables
t_arbre arbre

```

Et les correspondances entre les opérations abstraites et l'implémentation statique du type seraient les suivantes :

Type abstrait \Leftrightarrow Implémentation statique (hiérarchique)	
B=Arbre-vide	B.racine=0
racine(B)	B.racine
g(B)	B.noeuds[2*B.racine]
d(B)	B.noeuds[2*B.racine+1]
Contenu(racine(B))	B.noeuds[B.racine]

Parcours d'un arbre binaire

De nombreux algorithmes sur les arbres examinent systématiquement tous les noeuds d'un arbre pour y effectuer un traitement particulier. Cette opération s'appelle le parcours d'un arbre.

Il existe plusieurs formes de parcours d'arbre :

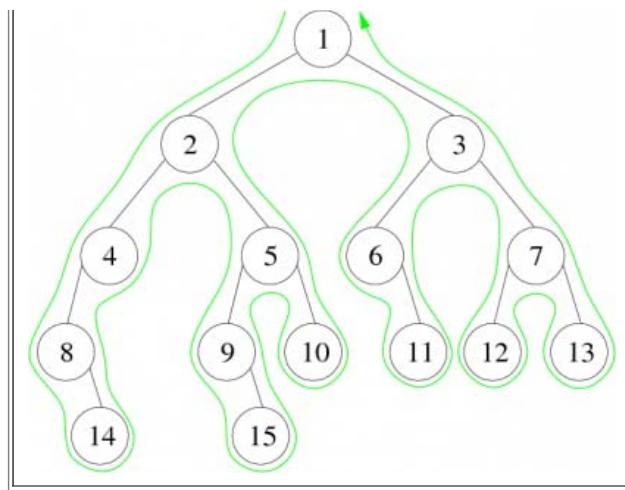
- le parcours en profondeur (Ooohh...).
- le parcours en largeur (Aaaahh...)

Le premier consiste à étudier les noeuds en descendant à chaque fois le plus loin possible dans l'arbre. Le deuxième consiste à passer en revue tous les noeuds de l'arbre niveau par niveau.

Parcours en profondeur

Parcours **par nature récursif**, celui-ci est une forme dérivée des sorties labyrinthiques. C'est à dire que l'on pose sa main gauche sur un mur, et que l'on avance en laissant celle-ci posée. Dans la plupart des cas (Franquin et ses idées noires vous en fourniront au moins un autre), nous arriverons fatallement à la sortie. Cette forme de parcours sur les arbres binaires s'appelle le parcours en profondeur main gauche.

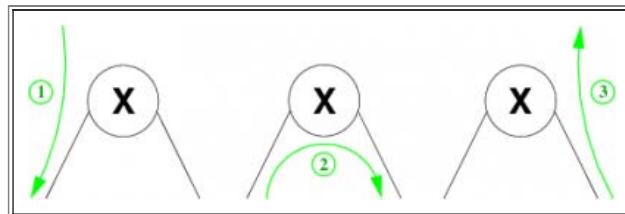
Figure 8. Parcours profondeur main gauche d'un arbre binaire



Sur le parcours de la figure 8, nous n'avons pas représenté les sous-arbres vides, mais ceux-ci doivent être pris en compte si l'on veut admettre que chaque noeud (y compris les feuilles) est rencontré plusieurs fois lors du parcours. Les ordres de rencontres induits par le parcours profondeur main gauche sont au nombre de 3 (cf figure 9) et sont appelés :

- (1) **Ordre préfixe** pour la descente vers le sous-arbre gauche,
- (2) **Ordre infixé** pour le passage du sous-arbre gauche au sous-arbre droit
- (3) **Ordre suffixe** pour la remontée depuis le sous-arbre droit.

Figure 9. Ordres induits par le parcours profondeur main gauche d'un arbre binaire



Algorithme de parcours profondeur main gauche

Lors de ce parcours, chaque noeud étant rencontré trois fois, nous pouvons affecter à chacun d'eux un traitement particulier. De plus, si l'on intègre les arbres vides, nous pouvons aussi y faire correspondre un traitement. Ce qui donne l'algorithme *abstrait* suivant, de parcours récursif d'un arbre :

```

Algorithme procédure parc_prof
Paramètres locaux
arbrebinaire b
Début
  si B=arbre-vide alors
    /* Traitement de fin */
  sinon
    /* Traitement préfixe */
    parc_prof(g(b))
    /* Traitement infixé */
    parc_prof(d(b))
    /* Traitement suffixe */
  fin si
Fin algorithme procédure parc_prof

```

Pour bien mettre en évidence les différences entre les trois ordres induits par ce parcours, nous allons remplacer à chaque fois chacun d'eux par l'ordre d'affichage suivant du noeud :

```
Ecrire(contenu(racine(b)))
```

En laissant les autres traitements vides, si l'on appliquait cet algorithme à l'arbre binaire de la figure 1, voilà ce qui serait obtenu à chaque fois :

- (1) **Ordre préfixe** : 1, 2, 4, 8, 14, 5, 9, 15, 10, 3, 6, 11, 7, 12, 13
- (2) **Ordre infixé** : 8, 14, 4, 2, 9, 15, 5, 10, 1, 6, 11, 3, 12, 7, 13

- (3) **Ordre suffixe** : 14, 8, 4, 15, 9, 10, 5, 2, 11, 6, 12, 13, 7, 3, 1

On constate que selon le type de parcours l'ordre des nœuds diffère (Bon d'accord : "Fer !").

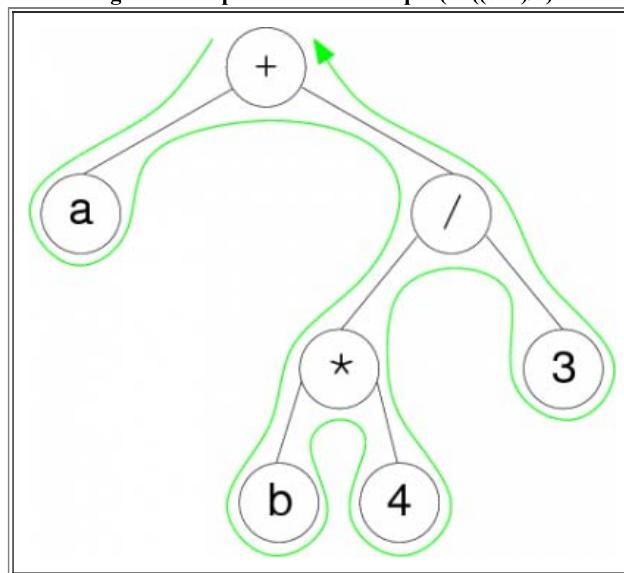
Remarques :

- L'ordre hiérarchique, (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) pour cet exemple, n'apparaît pas dans la mesure où celui-ci correspond à un parcours en largeur (par niveau) de l'arbre.
- Tous les traitements peuvent être utilisés en même temps pour, éventuellement, réaliser des choses diverses dans un ordre différent, ou alors parce que ces traitements sont complémentaires. Nous allons voir plus loin la nécessité d'utiliser plusieurs de ces traitements.

Parcours profondeur d'un arbre étiqueté représentant une expression arithmétique

Nous pouvons utiliser un arbre binaire pour représenter une expression arithmétique. Dans ce cas, les noeuds internes sont les opérateurs et les feuilles sont les opérandes. Ce qui, pour l'expression arithmétique ($a + ((b * 4) / 3)$), pourrait donner l'**arbre binaire étiqueté** de la figure 10 :

Figure 10. Expression arithmétique ($a + ((b * 4) / 3)$)



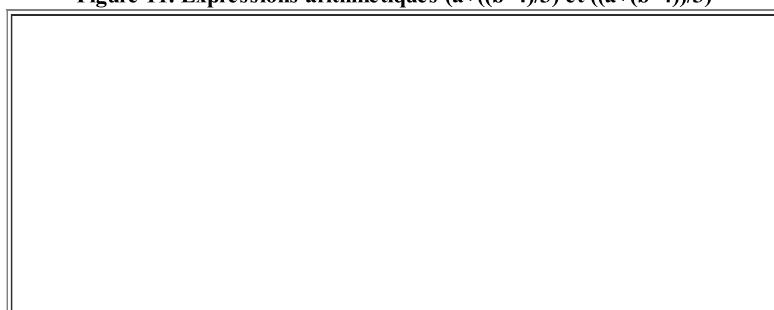
Remarque : Cette expression n'utilise que des opérateurs binaires (ayant deux opérandes), il est donc localement complet.

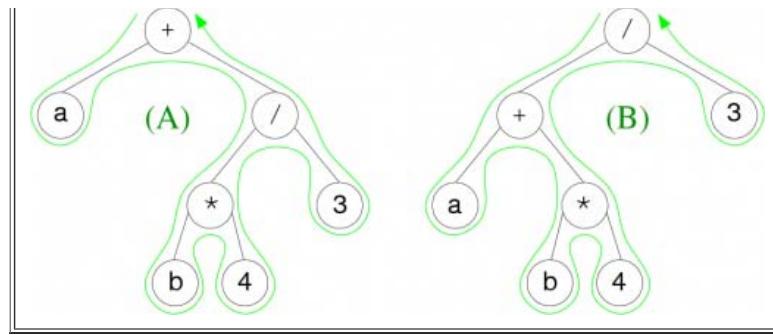
Pour l'arbre de la figure 10, le parcours selon les différents ordres donnerait :

- (1) **ordre préfixe** : + a * b 4 3
- (2) **ordre infixé** : a + b * 4 / 3
- (3) **ordre suffixe** : a b 4 * 3 / +

On constate que l'ordre préfixe et l'ordre suffixe sont non ambigus, le premier correspondant à une **notation polonaise** et le deuxième à une **notation polonaise inversée (rpn)**. Ce n'est malheureusement pas le cas de l'ordre infixé pour qui cette série pourrait correspondre à un tout autre arbre comme le montre la figure 11.

Figure 11. Expressions arithmétiques ($a + ((b * 4) / 3)$ et $((a + (b * 4)) / 3)$)





En effet les notations en polonaise ou en polonaise inversée font précéder ou suivre les opérateurs de leurs opérandes. Or pour la notation en ordre infixe, nous ne savons pas si $a+b*4/3$ correspond à $a+((b*4)/3)$ ou à $(a+(b*4))/3$. En fait cette ambiguïté peut être facilement levée à l'aide de parenthèses qui forcent alors la priorité des opérateurs.

Il suffit alors de modifier l'algorithme de parcours en profondeur main gauche donné précédemment de la manière suivante :

```

Algorithme procédure parc_prof_infixe
Paramètres locaux
    arbrebinnaire b
Début
    si g(b)=arbre-vide et d(b)=arbre-vide alors /* b est une feuille */
        écrire(contenu(racine(b)))
        /* Traitement de fin */
    sinon
        écrire('(')
        parc_prof(g(b))
        écrire(contenu(racine(b)))
        parc_prof(d(b))
        écrire(')')
        /* Traitement préfixe */
        /* Traitement infixé */
        /* Traitement suffixe */
    fin si
Fin algorithme procédure parc_prof_infixe

```

Ce qui pour les arbres donnés en figure 11 donnerait les parcours infixes non ambigus suivants :

- **Figure 11(A) :** $(a + ((b * 4) / 3))$
- **Figure 10(B) :** $((a + (b * 4)) / 3)$

Remarque : Dans l'algorithme, nous aurions pu modifier le test qui vérifie si le noeud sur lequel nous sommes est une feuille. Pour cela, il aurait fallu faire une extension au type abstrait ArbreBinaire en créant une nouvelle opération comme suit :

```

opérations
feuille : arbrebinnaire → booléen

axiomes
b ≠ arbrevide & g(b)=arbrevide & d(b)=arbrevide ⇒ feuille(b)=vrai

```

Le début de l'algorithme serait alors :

```

Algorithme procédure parc_prof_infixe
Paramètres locaux
    arbrebinnaire b
Début
    si feuille(b) alors /* b est une feuille */

```

Parcours en largeur

Ce parcours consiste à suivre, à partir de la racine de l'arbre, les deux fils de celle-ci, puis à passer aux deux fils du 1er fils, puis aux deux fils du 2ème fils, etc. Le parcours se fait, en fait, par distance (*hauteur dans ce cas*), c'est à dire que l'on parcourt d'abord tous les noeuds se trouvant à une distance de 1 de la racine, puis tous ceux qui se trouvent à une distance de 2 et ainsi de suite...

Remarque : Ce parcours aussi qualifié de **hiérarchique** est par nature itératif.

L'algorithme de parcours largeur utilise une File pour mémoriser les deux fils de chaque noeud rencontré. Ce qui permet à la hauteur suivante de les récupérer dans l'ordre de rencontre; Cela donne l'algorithme suivant :

```

Algorithme procédure parc_larg
algo.infoprepa.epita.fr/index.php?title=Epita:Algo:Cours:Info-Sup:Structures_arborescentes&printable=yes

```

```

Paramètres locaux
arbrebinnaire b
Variables
entier i
file f
Début
f ← filevide
f ← enfiler(f,b)
tant que non(estvide(f)) faire
b ← premier(f)
f ← defiler(f)
écrire(contenu(racine(b)))
si g(b)>arbre-vide alors
f ← enfiler(f,g(b))
fin si
si d(b)>arbre-vide alors
f ← enfiler(f,d(b))
fin si
fin tant que
Fin Algorithmé procédure parc_larg

```

En utilisant l'arbre général de la Figure 8, nous obtenons l'affichage des noeuds dans l'ordre suivant :

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

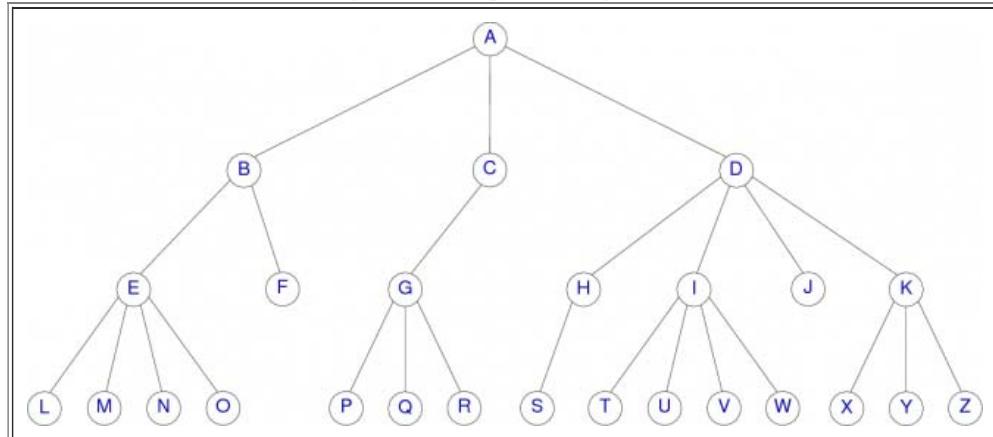
Remarques :

- Cet algorithme effectue le parcours largeur d'un arbre binaire quelconque **non vide**.
- La complexité est la même que pour le parcours en profondeur.

Les arbres généraux

Un arbre général ou arbre est une structure arborescente où le nombre de fils n'est pas limité à deux. La définition récursive d'un arbre général est $A = < o, A_1, \dots, A_n >$ où A est la donnée d'une racine o et d'une liste finie, éventuellement vide (si $n=0$), d'arbres disjoints $< A_1, \dots, A_n >$. On appelle cette liste une forêt (Alors, hein ? On ne me croyait pas !). On obtient donc un arbre en ajoutant une racine à une forêt (Ca tombe sous le sens).

Figure 12. Exemple d'arbre général.



Le type *Arbre général*

Le type abstrait d'un arbre général est le suivant :

```

types
arbre, forêt
utilise
noeud, entier, élément

opérations
cons : noeud x forêt → arbre
racine : arbre → noeud
listearbre : arbre → forêt
forêtvide : → forêt
insérer : forêt x entier x arbre → forêt
supprimer : forêt x entier → forêt
ième : forêt x entier → arbre

```

```

nbarbres : forêt → entier
contenu : noeud → élément

préconditions

insérer(F,i,A) est-défini-ssi  $1 \leq i \leq 1 + \text{nbarbres}(F)$ 
supprimer(F,i) est-défini-ssi  $F \neq \text{forêt-vide} \& 1 \leq i \leq \text{nbarbres}(F)$ 
ième(F,i) est-défini-ssi  $F \neq \text{forêt-vide} \& 1 \leq i \leq \text{nbarbres}(F)$ 

axiomes

racine(cons(o,F))=o
listearbre(cons(o,F))=F
nbarbres(forêt-vide)=0
nbarbres(insérer(F,i,A))=nbarbres(F') + 1
nbarbres(supprimer(F,i))=nbarbres(F') - 1
 $1 \leq i < k \Rightarrow \text{ième}(\text{insérer}(F,k,A), i) = \text{ième}(F, i)$ 
 $k = i \Rightarrow \text{ième}(\text{insérer}(F,k,A), i) = A$ 
 $k < i \leq \text{nbarbres}(F)+1 \Rightarrow \text{ième}(\text{insérer}(F,k,A), i) = \text{ième}(F, i-1)$ 
 $1 \leq i < k \Rightarrow \text{ième}(\text{supprimer}(F,k), i) = \text{ième}(F, i)$ 
 $k \leq i \leq \text{nbarbres}(F)-1 \Rightarrow \text{ième}(\text{supprimer}(F,k), i) = \text{ième}(F, i+1)$ 

avec
noeud o
arbre A
forêt F
entier i,k

```

Remarques :

- La notion de « gauche-droite » propre aux arbres binaires disparaît. Cela dit, le vocabulaire employé pour les arbres généraux reste le même hormis pour tout ce qui fait appel à cette notion (de généralité). Par exemple, on ne parle plus de fils gauche-fils droit, mais de 1er fils, 2ème fils, etc. et de dernier fils.
- Les mesures sur les arbres sont conservées (hauteur, longueur de cheminement, taille, etc.).

Représentation des arbres généraux

Il existe plusieurs formes de représentations des arbres généraux. Les plus intuitives sont la représentation sous forme de listes chaînées et n-uplet. Une autre forme très usitée est la représentation sous forme d'arbres binaires.

Représentation sous forme statique-dynamique

Dans ce cas de figure, l'ensemble des noeuds est représenté dans un tableau à une dimension (vecteur) surdimensionné pour pouvoir y ajouter des noeuds le cas échéant. Chaque élément est un enregistrement qui contient l'étiquette du noeud plus un pointeur sur sa liste de fils. La liste de fils est composée d'enregistrements contenant un pointeur sur le type d'élément constituant le tableau et un pointeur suivant (comme pour les listes d'adjacences de graphe). Dans ce cas, sa description algorithmique serait :

```

Constantes

Nbmax = 27

Types

t_element = ... /* Définition du type des éléments */
t_pfil = ↑t_fils /* Définition du pointeur sur noeud fils (liste) */

t_noeud = enregistrement /* Définition du type noeud */
    t_element elt
    t_pfil premierfils
fin enregistrement t_noeud

t_fils = enregistrement /* Définition du type t_fils */
    entier noeudfils
    t_pfil filssuiv
fin enregistrement t_fils

t_arbre = Nbmax t_noeud /* Définition du type arbre (tableau) */

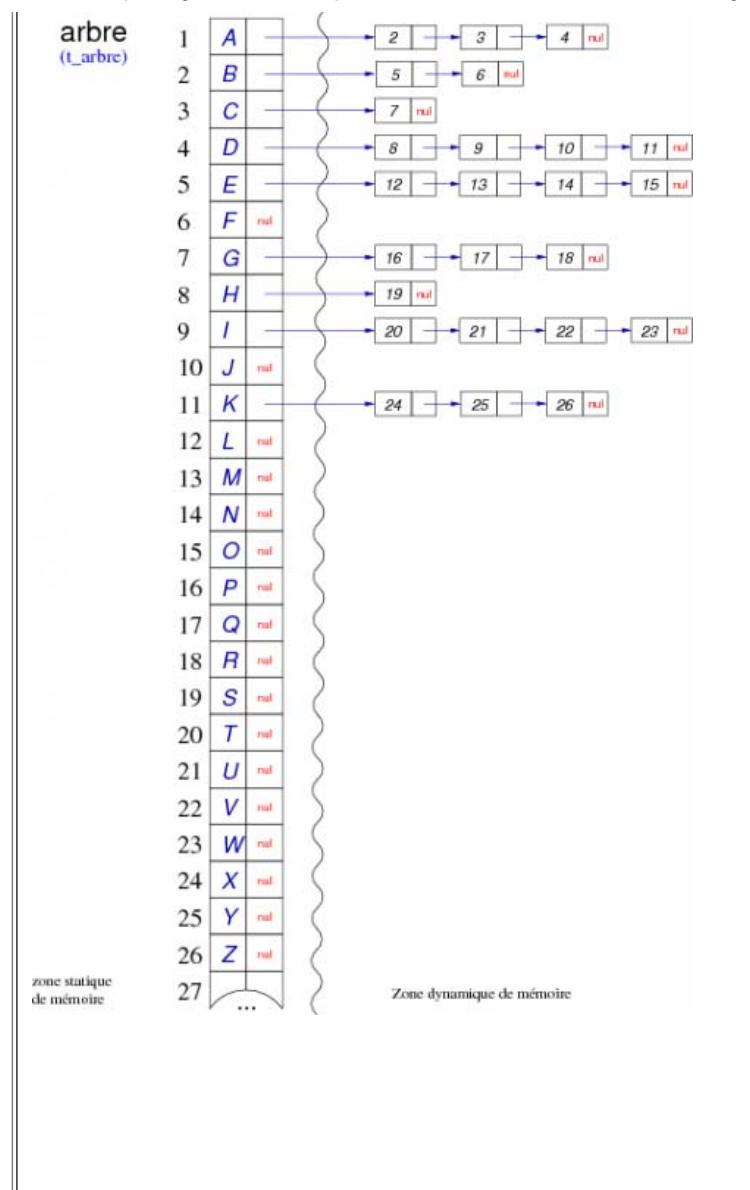
Variables

t_arbre arbre

```

En utilisant l'arbre de la figure 12, cela donnerait :

Figure 13. Représentation statique-Dynamique de l'arbre de la figure 12.



Représentation sous forme dynamique

Remarque : Une possibilité est de transformer cette représentation totalement en dynamique en remplaçant le tableau statique par une liste chaînée. Dans ce cas, les noeuds sont reliés entre eux par un lien dynamique (pointeur) et de ce fait, l'entier **noeud fils** (indice du noeud fils dans le tableau) se trouve, dans l'enregistrement **t_fils**, remplacé par un pointeur sur l'adresse du noeud.

Ce qui donnerait la déclaration algorithmique suivante :

```

Types

t_element = ...
t_pnoeud = ↑t_noeud
t_pfils = ↑t_fils

/* Définition du type des éléments */
/* Définition du type pointeur sur noeud */
/* Définition du type pointeur sur noeud fils */

t_noeud = enregistrement
    t_element elt
    t_pnoeud noeudsuiv, noeuprec
    t_pfils premierfils
fin enregistrement t_noeud

t_fils = enregistrement
    t_pnoeud noeudfils
    t_pfils filssuiv
fin enregistrement t_fils

t_arbre = t_pnoeud
/* Définition du type arbre (pointeur sur un noeud) */

Variables

t_arbre arbre

```

Pour visualiser cette structure, il suffit d'aller voir la représentation des graphes sous formes de listes d'adjacences avec ensemble de sommet dynamique.

Représentation sous forme d'arbre binaire

Le moyen de représenter les arbres généraux sous forme d'arbres binaires est d'utiliser le lien gauche comme premier fils et le lien droit comme frère droit. Cette représentation présente plusieurs avantages :

- Il y a un noeud par élément ni plus ni moins (*bijection premier fils-frère droit*).
- Nous pouvons utiliser les algorithmes sur les arbres binaires (parcours, ajout de noeud, etc.) qui sont très simples à mettre en place.

Remarque : Il semble évident bien sûr que c'est l'implémentation dynamique de l'arbre binaire qui sera retenue.

La déclaration algorithmique serait alors la suivante :

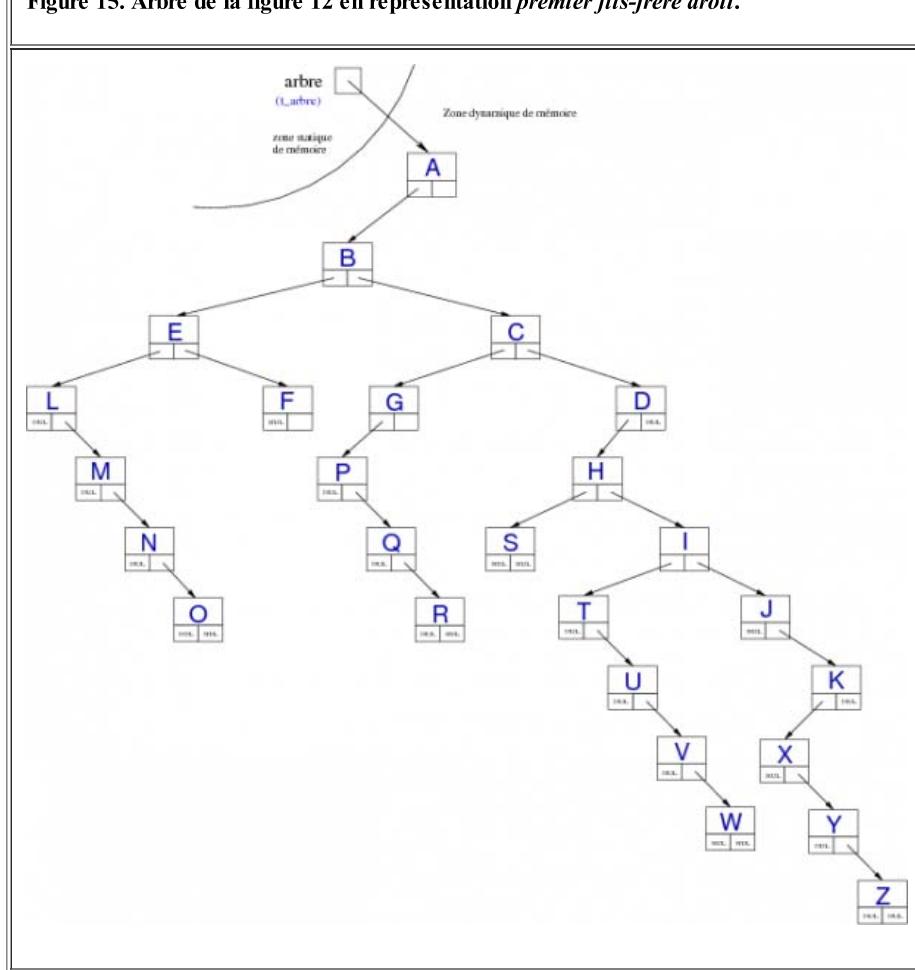
```
Types
t_element = ...
t_arbre = ↑t_noeud
t_noeud = enregistrement
    t_element elt
    t_arbre premierfils, frerdroit
fin enregistrement t_noeud
```

Variables

```
t_arbre arbre
```

Ce qui correspondrait à la structure suivante :

Figure 15. Arbre de la figure 12 en représentation *premier fils-frère droit*.



Travailler à l'aide de cet arbre est très simple. Par exemple pour connaître les fils d'un noeud (dans l'arbre général), il suffit, à partir de son fils droit (dans l'arbre binaire) de parcourir le bord droit (toujours dans l'arbre binaire). De la même manière, calculer la hauteur de l'arbre général se fait dans l'arbre binaire en ne considérant que la hauteur des fils gauches (qui représente les premiers fils dans l'arbre général) et pas celle des fils droits (qui représentent les frères situés à la même hauteur).

Remarques :

- L'arbre obtenu n'est pas du tout équilibré.
- Le parcours préfixe et le parcours symétrique de cet arbre binaire donne respectivement le même ordre de rencontre des noeuds que le parcours

préfixe et le parcours postfixe de l'arbre général que celui-ci représente.

Représentation sous forme de N-uplet

SI l'on connaît le nombre maximum de fils que peut avoir un noeud, il y a possibilité de représenter l'arbre sous forme de **n-uplet**. Cette représentation est une extension de la représentation dynamique des arbres binaires. Là, le nombre de fils n'est pas deux, mais celui que l'on s'est fixé (*précisé à l'aide d'une constante*). En supposant qu'il y ait cinq fils maximum, la déclaration algorithmique serait la suivante :

```
Constantes
Nbmaxfils = 5          /* Nombre maximum de fils */

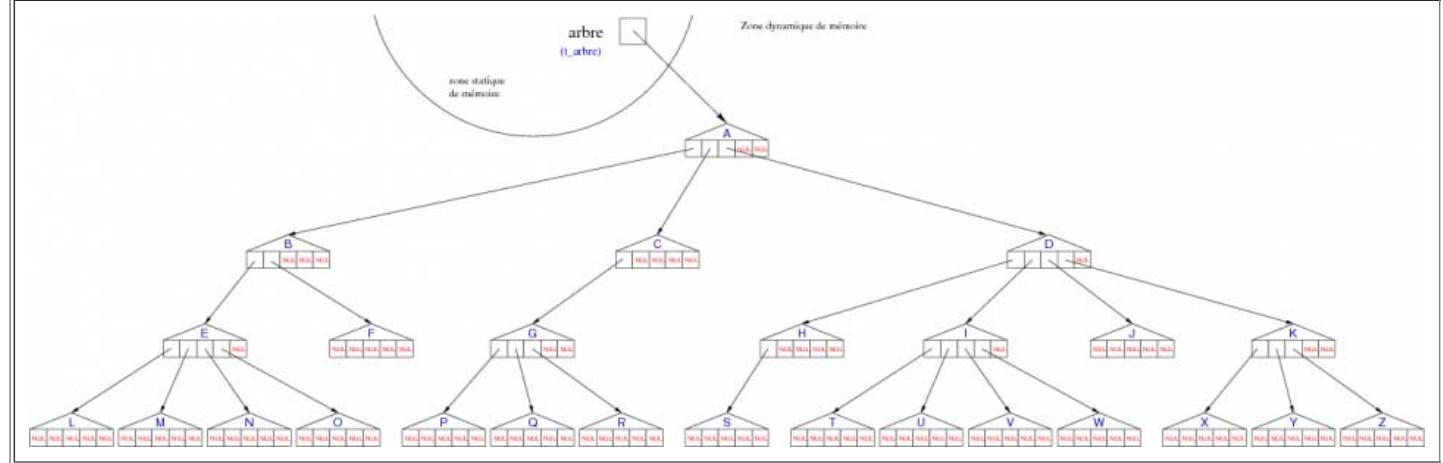
Types
t_element = ...          /* Définition du type des éléments */
t_noeud = ↑t_noeud      /* Définition de l'arbre */
t_tabfils = Nbmaxfils t_arbre /* Définition du tableau de pointeurs sur fils */

t_noeud = enregistrement    /* Définition du type noeud */
  t_element elt
  t_tabfils fils
fin enregistrement t_noeud

Variables
t_arbre arbre
```

Et sa représentation graphique en utilisant l'arbre de la figure 12 serait :

Figure 16. Arbre de la figure 12 en représentation n-uplet.



Parcours d'un arbre général

Comme pour les arbres binaires, de nombreux algorithmes sur les arbres examinent systématiquement tous les noeuds d'un arbre pour y effectuer un traitement particulier. Cette opération s'appelle le parcours d'un arbre.

Il existe plusieurs formes de parcours d'arbre :

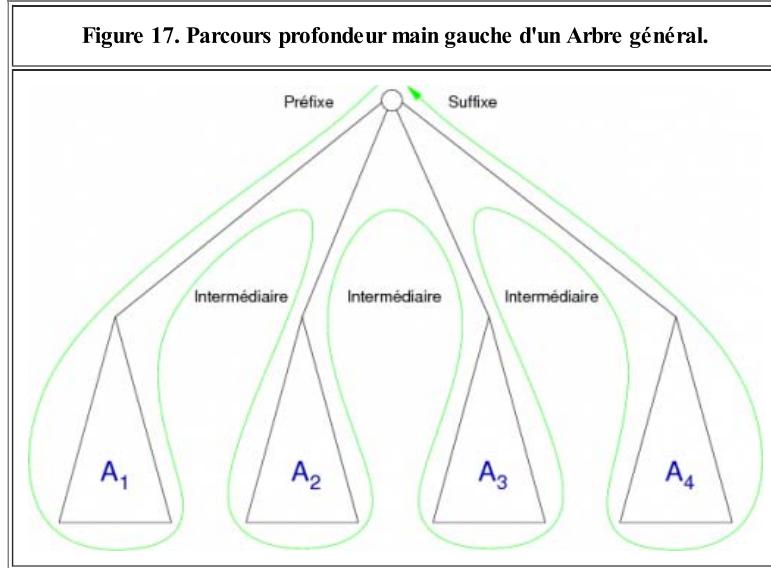
- le parcours en profondeur (Ooohh...).
- le parcours en largeur (Aaaahh...)

Le premier, **par nature récursif**, consiste à étudier les noeuds en descendant à chaque fois le plus loin possible dans l'arbre. Le second, **par nature itératif**, consiste à passer en revue tous les noeuds de l'arbre niveau par niveau.

Parcours en profondeur

Pour parcourir un arbre général, on peut réaliser une extension du parcours en profondeur main gauche des arbres binaires. La différence réside dans le fait qu'il n'y a pas de traitement infixé, mais un traitement intermédiaire à chaque passage d'un fils à un autre. En d'autres termes, un noeud est rencontré son nombre de fils plus une fois : la descente sur le premier fils, les "nombres de fils moins un" passage intermédiaire d'un fils à l'autre et enfin, la remontée depuis le dernier fils (C'est pas clair Junior ?). L'algorithme employé pour ce parcours est celui du parcours d'arbre binaire « légèrement modifié ». En effet, il faut placer les traitements à l'intérieur d'une boucle dont le nombre d'itérations repose sur le nombre de fils que possède l'arbre parcouru.

Pour illustrer ces propos, si l'on définit un arbre général **A=< o, A1, A2, A3, A4 >**, nous obtiendrons le parcours de la figure 17 (ressemblant à s'y méprendre à une radiographie de la main de Mickey mettant en évidence la présence d'un exo-squelette :D):



Plus sérieusement, on peut constater, sur ce parcours, que le noeud racine est visité cinq fois ; une fois en descendant vers **A1** (*traitement préfixe*), trois fois en passant de **A1** à **A2**, **A2** à **A3** et **A3** à **A4** (*traitements intermédiaires*) et enfin en remontant de **A4** (*traitement suffixe*).

Et là Junior, tu crois qu'ils ont compris ?

L'algorithme abstrait du parcours en profondeur d'un arbre général est alors :

```

Algorithme procédure parc_prof
Paramètres locaux
    arbre a
Variables
    entier i, nbfil
Début
    nbfil ← nbarbres(listearbre(a))
    si feuille(a) alors /* b est une feuille (nbfil=0) */
        /* Traitement de terminaison */
    sinon
        /* Traitement préfixe */
        pour i ← 1 jusqu'à nbfil-1 faire
            parc_prof(ième(listearbre(a),i))
            /* Traitement intermédiaire */
        fin pour
        parcours(ième(listearbre(a),nbfil))
        /* Traitement suffixe */
    fin si
Fin algorithme procédure parc_prof

```

Remarques :

- Nous utilisons une extension présentée lors du parcours d'arbre binaire à savoir : **feuille**. Cette opération devrait bien sûr être définie comme extension du type abstrait **arbre général**.
- Le nombre de traitements possibles sur un noeud ne se limite pas à trois, mais peut être réellement égal au nombre de fils plus un. En effet, le traitement intermédiaire peut varier selon la valeur de l'indice de boucle et par conséquent renvoyer à un traitement particulier à chaque fois.

Par exemple, en utilisant l'arbre général de la figure 12, nous pourrions lister les noeuds de la manière suivante :

```
(A (B (E (L) (M) (N) (O)) (F)) (C (G (P) (Q) (R))) (D (H (S)) (I (T) (U) (V) (W)) (J) (K (X) (Y) (Z))))
```

en utilisant l'algorithme ci-dessous :

```

Algorithme procédure parc_prof_lisp
Paramètres locaux
    arbre a
Variables
    entier i, nbfil
Début
    nbfil ← nbarbres(listearbre(a))
    si feuille(a) alors /* b est une feuille (nbfil=0) */
        écrire('(',contenu(racine(a)),')')
    sinon
        écrire('(',contenu(racine(a)))
        pour i ← 1 jusqu'à nbfil-1 faire
            parc_prof(ième(listearbre(a),i))
        fin pour
        parcours(ième(listearbre(a),nbfil))
        écrire(')')
    fin si

```

```
Fin algorithme procédure parc_prof_lisp
```

qui pourrait être optimisé de la manière suivante :

```
Algorithme procédure parc_prof_lisp
Paramètres locaux
    arbre a
Variables
    entier i, nbfil
Début
    nbfil ← nbarbres(listearbre(a))
    écrire('(' , contenu(racine(a)))
    si non(feuille(a)) alors /* b n'est pas une feuille (nbfil=0) */
        pour i ← 1 jusqu'à nbfil-1 faire
            parc_prof(ième(listearbre(a),i))
        fin pour
        parcours(ième(listearbre(a),nbfil))
    fin si
    écrire(')')
Fin algorithme procédure parc_prof_lisp
```

Parcours en largeur

Ce parcours consiste à suivre, à partir de la racine de l'arbre, tous les fils de celle-ci, puis à passer à tous les fils du 1er fils, puis à tous les fils du 2ème fils, etc. Le parcours se fait, en fait, par distance (*hauteur dans ce cas*), c'est à dire que l'on parcourt d'abord tous les noeuds se trouvant à une distance de 1 de la racine, puis tous ceux qui se trouvent à une distance de 2 et ainsi de suite...

*Remarque : Ce parcours aussi qualifié de **hiérarchique** est par nature itératif.*

L'algorithme de parcours largeur utilise une File pour mémoriser les descendants directs de chaque noeud rencontré. Ce qui permet à la hauteur suivante de les récupérer dans l'ordre de rencontre; Cela donne l'algorithme suivant :

```
Algorithme procédure parc_larg
Paramètres locaux
    arbre a
Variables
    entier i
    file f
                    /* f stocke les arbres */
Début
    f ← filevide
    f ← enfiler(f,a)
                    /* stockage de l'arbre */
    tant que non(estvide(f)) faire
        a ← premier(f)
                    /* récupération du 1er arbre stocké */
        f ← defiler(f)
                    /* libérer la file de cet arbre */
        écrire(contenu(racine(a)))
                    /* affichage du noeud racine de l'arbre récupéré */
        pour i ← 1 jusqu'à nbarbres(listearbre(a)) faire
            f ← enfiler(f,ième(listearbre(a),i))
                    /* stockage du ième sous-arbre de a */
        fin pour
    fin tant que
Fin Algorithme procédure parc_larg
```

En utilisant l'arbre général de la Figure 12, nous obtenons l'affichage des noeuds dans l'ordre suivant :

```
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
```

Remarques : La complexité est la même que pour le parcours en profondeur.

(Christophe "krisboul" Boullay)

Récupérée de « http://algo.infoprepa.epita.fr/index.php?title=Epita:Algo:Cours:Info-Sup:Structures_arborescentes&oldid=2470 »

- Dernière modification de cette page le 27 décembre 2012 à 17:28.