Chapitre 7
Structures arborescentes

Un arbre est un ensemble de *nœuds*, organisés de façon hiérarchique, à partir d'un nœud distingué, appelé racine. La structure d'arbre est l'une des plus importantes et des plus spécifiques de l'informatique : par exemple, c'est sous forme d'arbre que sont organisés les fichiers dans des systèmes d'exploitation tels que UNIX ou MULTICS; c'est aussi sous forme d'arbres que sont représentés les programmes traités par un compilateur...

Une propriété intrinsèque de la structure d'arbre est la récursivité, et les définitions des caractéristiques des arbres, aussi bien que les algorithmes qui manipulent des arbres s'écrivent très naturellement de manière récursive.

1. Arbres binaires

Examinons tout d'abord quelques exemples simples représentés par des arbres binaires :

- les résultats d'un tournoi de tennis : la figure 1 montre qu'au premier tour, Marc a battu François, Jean a battu Jules et Luc a battu Pierre; au deuxième tour Jean a battu Marc, et Paul a battu Luc; et Jean a gagné en finale contre Paul.

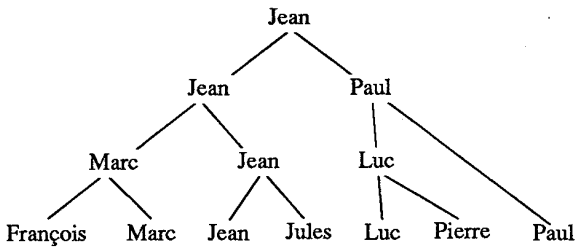

Figure 1. Tournoi de tennis.

– le pedigree d'un cheval de course : la figure 2 décrit le pedigree du cheval Zoe; son père est Tonnerre et sa mère Belle, la mère de Belle s'appelle Rose et son père Eclair...

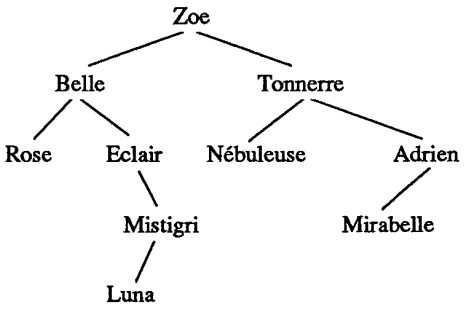

Figure 2. Pedigree de Zoe.

– une expression arithmétique dans laquelle tous les opérateurs sont binaires; l'arbre de la figure 3 code l'expression :

$$(x - (2 * y)) + ((x + (y/z)) * 3)$$

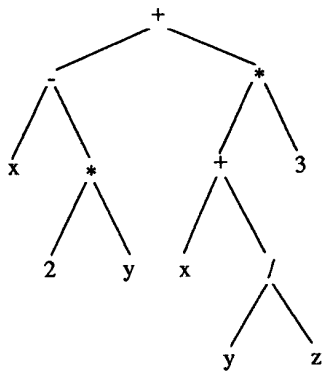

Figure 3. Expression arithmétique.

La structure d'arbre binaire est utilisée dans de très nombreuses applications informatiques; de plus les arbres binaires permettent de représenter les arbres plus généraux présentés au paragraphe 2.

1.1. Définitions

1.1.1. Le type arbre binaire

Définition 1 : Un ***arbre binaire*** est soit vide (noté Ø), soit de la forme $B =< o, B_1, B_2 >$, où B_1 et B_2 sont des arbres binaires disjoints et 'o' est un nœud appelé ***racine***.

Cette définition est récursive; elle peut s'écrire sous la forme de l'équation :

$$\mathbb{B} = Ø + < o,\ \mathbb{B},\ \mathbb{B} >$$

où $\mathbb{B}$ représente l'ensemble des arbres binaires, Ø représente l'arbre vide, et o désigne un nœud.

Remarque : Il est important de noter la **non-symétrie** gauche-droite des arbres binaires : l'arbre $< o,\ < o,\ Ø,\ Ø >,\ Ø >$ et l'arbre $< o,\ Ø,\ < o,\ Ø,\ Ø >>$ sont des arbres binaires différents. Pour pouvoir repérer les nœuds dans un arbre, il faut associer à chacun d'entre eux un nom différent. Sur la figure 4, on a représenté un arbre binaire en inscrivant auprès de chaque nœud une lettre indicée qui le désigne.

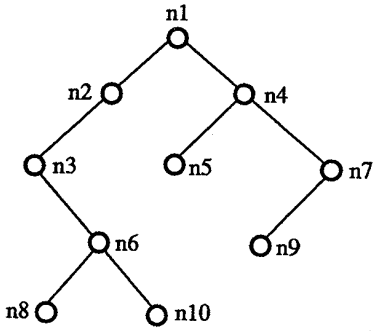

Figure 4. Représentation graphique d'un arbre binaire.

La signature du type abstrait arbre binaire est :

sorte Arbre
utilise Nœud, Elément
opérations

arbre-vide	:	→ Arbre
$< -, -, - >$	:	Nœud × Arbre × Arbre → Arbre
racine	:	Arbre → Nœud
g	:	Arbre → Arbre
d	:	Arbre → Arbre
contenu	:	Nœud → Elément

Etant donnés B_1 et B_2 des variables de sorte Arbre, et o une variable de sorte Nœud, on a les préconditions et axiomes suivants :

préconditions

racine(B_1) **est-défini-ssi** $B_1 \neq$ *arbre-vide*
$g(B_1)$ **est-défini-ssi** $B_1 \neq$ *arbre-vide*
$d(B_1)$ **est-défini-ssi** $B_1 \neq$ *arbre-vide*

axiomes

racine$(< o,\ B_1,\ B_2 >) = o$
$g(< o,\ B_1,\ B_2 >) = B_1$
$d(< o,\ B_1,\ B_2 >) = B_2$

L'opération *contenu* permet d'associer à chaque Nœud de l'Arbre une information de type Elément.

Un arbre dont les nœuds contiennent des éléments est dit ***arbre étiqueté.***

Si $B =< o,\ B_1,\ B_2 >$ est un arbre étiqueté dont la racine contient l'élément r, on note, abusivement, $B =< r,\ B_1,\ B_2 >$.

Le vocabulaire concernant les arbres informatiques est souvent emprunté à la botanique ou à la généalogie; étant donné un arbre $B =< o,\ B_1,\ B_2 >$:

• *'o'* est la ***racine*** de B.

• B_1 est le ***sous-arbre gauche*** de la racine de B, (ou, plus simplement, le sous-arbre gauche de B), et B_2 est son ***sous-arbre droit.***

• On dit que C est un ***sous-arbre*** de B si, et seulement si : $C = B$, ou $C = B_1$, ou $C = B_2$, ou C est un sous-arbre de B_1 ou de B_2.

• On appelle ***fils gauche*** (respectivement ***fils droit***) d'un nœud la racine de son sous-arbre gauche (respectivement sous-arbre droit), et l'on dit qu'il y a un ***lien gauche*** (respectivement droit) entre la racine et son fils gauche (respectivement fils droit).

• Si un nœud n_i a pour fils gauche (respectivement droit) un nœud n_j, on dit que n_i est le ***père*** de n_j (remarquons que d'après la définition 1 chaque nœud n'a qu'un seul père).

• Deux nœuds qui ont le même père sont dits ***frères.***

• Le nœud n_i est un ***ascendant*** ou un ***ancêtre*** du nœud n_j si, et seulement si, n_i est le père de n_j, ou un ascendant du père de n_j ; n_i est un ***descendant*** de n_j si, et seulement si n_i est fils de n_j, ou n_i est un descendant d'un fils de n_j.

• Tous les nœuds d'un arbre binaire ont au plus deux fils :
  – un nœud qui a deux fils est appelé ***nœud interne*** ou ***point double***,

- un nœud qui a seulement un fils gauche (respectivement droit) est dit ***point simple à gauche*** (resp. ***point simple à droite***), ou ***nœud interne au sens large***,
- un nœud sans fils est appelé ***nœud externe*** ou ***feuille***.

• On appelle ***branche*** de l'arbre B tout chemin (un chemin est une suite de nœuds consécutifs) de la racine à une feuille de B : un arbre a donc autant de branches que de feuilles.

• On appelle ***bord gauche*** (respectivement droit) de l'arbre B le chemin obtenu à partir de la racine en ne suivant que des liens gauches (respectivement droits) dans B.

1.1.2. Mesures sur les arbres

On introduit maintenant quelques opérations sur les arbres, à valeurs dans les entiers (ou dans les réels). Ces opérations seront utiles pour évaluer la complexité des algorithmes sur les arbres.

• La ***taille*** d'un arbre est le nombre de ses nœuds; on définit récursivement l'opération *taille* par :

taille(arbre-vide) = 0, et
taille$(< o,\ B_1,\ B_2 >) = 1 +$ *taille*$(B_1) +$ *taille*(B_2).

• La ***hauteur d'un nœud*** (on dit aussi ***profondeur*** ou ***niveau***) est définie récursivement de la façon suivante, étant donné x un nœud de B,

$h(x) = 0$ si x est la racine de B, et
$h(x) = 1 + h(y)$ si y est le père de x.

Ainsi la hauteur d'un nœud est comptée par le nombre de liens sur l'unique chemin de la racine à ce nœud.

• La ***hauteur*** ou ***profondeur*** d'un arbre B est :

$h(B) = \max\{h(x); x$ nœud de $B\}$

• La ***longueur de cheminement*** d'un arbre B est :

$LC(B) = \sum h(x)$, la somme étant prise sur tous les nœuds x de B.

Notons que l'on peut donner aussi une définition récursive de la hauteur et de la longueur de cheminement d'un arbre (cf. exercices).

• Dans la longueur de cheminement d'un arbre, on peut distinguer la contribution des feuilles de celle des autres nœuds :

- la ***longueur de cheminement externe*** d'un arbre B est
  $LCE(B) = \sum h(f)$, la somme étant prise sur toutes les feuilles f de B.
- la ***longueur de cheminement interne*** (au sens large) d'un arbre B est
  $LCI(B) = \sum h(x)$, la somme étant prise sur tous les nœuds internes (au sens large) x de B.

Et l'on a évidemment la relation $LC(B) = LCE(B) + LCI(B)$

Les quantités *longueurs de cheminement*, sont des sommes de profondeurs de nœuds ; en divisant chacune de ces quantités par le nombre de nœuds intervenant dans la somme, on obtient une notion de profondeur moyenne d'un nœud (quelconque, externe, ou interne au sens large).

Par exemple la ***profondeur moyenne externe*** d'un arbre B ayant f feuilles est

$$PE(B) = \frac{1}{f} LCE(B)$$

Il est important de ne pas confondre $PE(B)$ qui est la profondeur moyenne d'une feuille de B, et $h(B)$ qui est la profondeur maximale des feuilles de B.

Exemple : Sur l'arbre B de la figure 4, la racine n_1 a pour fils gauche n_2 et pour fils droit n_4 ; n_8 et n_{10} sont frères et n_6 est leur père ; les points doubles sont n_1, n_4 et n_6 ; n_8, n_9 et n_{10} sont des feuilles ; n_2 et n_7 sont des points simples à gauche, et n_3 est un point simple à droite ; le bord gauche est (n_1, n_2, n_3) ; le bord droit est (n_1, n_4, n_7) ; $(n_1, n_2, n_3, n_6, n_8)$ et $(n_1, n_2, n_3, n_6, n_{10})$ sont deux branches de l'arbre. Les hauteurs des nœuds sont $h(n_1) = 0$, $h(n_2) = h(n_4) = 1$, $h(n_3) = h(n_5) = h(n_7) = 2$, $h(n_6) = h(n_9) = 3$ et $h(n_8) = h(n_{10}) = 4$.

Caractéristiques de l'arbre : $h(B) = 4$; $LC(B) = 21$; $LCI(B) = 9$; $LCE(B) = 13$; $PE(B) = 13/4 = 3.24$.

1.1.3. Arbres binaires particuliers

On définit ici certaines formes particulières d'arbres binaires qui jouent un rôle important par la suite : les formes extrêmes d'arbres binaires (dégénérés , complets ou parfaits), et les arbres binaires sans points simples (localement complets).

• Un arbre binaire ***dégénéré*** ou ***filiforme*** est un arbre formé uniquement de points simples (voir figure 5.a).

• Un arbre binaire est dit ***complet*** s'il contient 1 nœud au niveau 0, 2 nœuds au niveau 1, 4 nœuds au niveau 2,..., 2^h nœuds au niveau h (voir figure 5.b). Dans un tel arbre on dit aussi que chaque *niveau est complètement rempli.*

Le nombre total de nœuds d'un arbre complet de hauteur h est donc $n = 1 + 2 + ... + 2^h = 2^{h+1} - 1$.

La définition suivante élargit la notion d'arbre complet aux arbres de taille quelconque.

• Un arbre binaire ***parfait*** est un arbre binaire dont tous les niveaux sont complètement remplis, sauf éventuellement le dernier niveau, et dans ce cas les nœuds (feuilles) du dernier niveau sont groupés le plus à gauche possible.

Par exemple dans la figure 6, l'arbre (a) est parfait mais les arbres (b) et (c) ne sont pas parfaits.

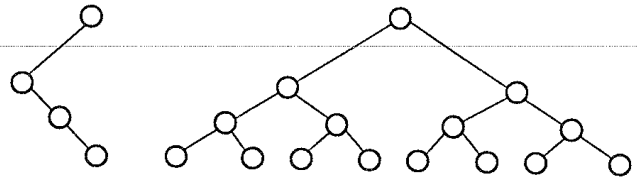

Figure 5. Arbres binaires extrêmes (a) dégénéré, (b) complet.

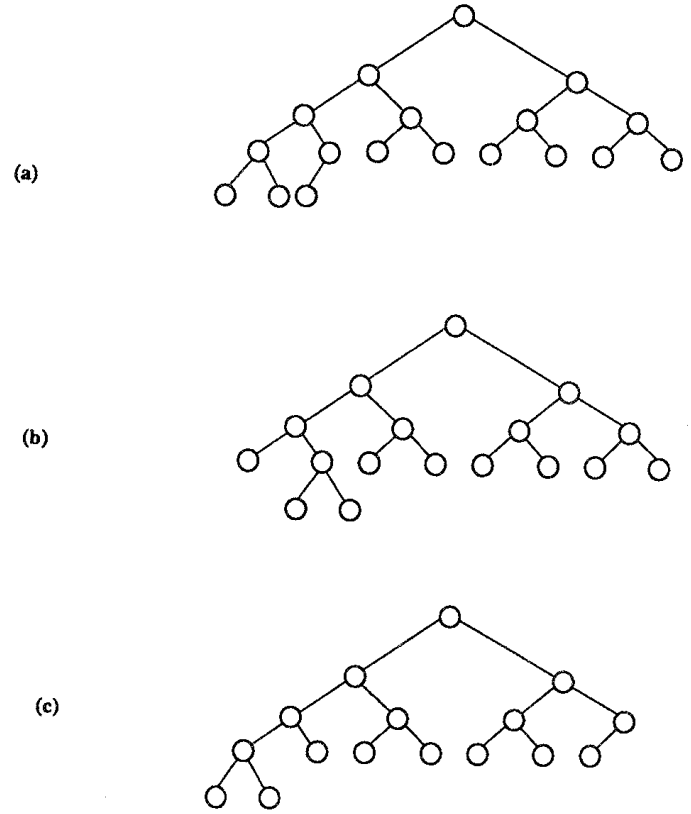

Figure 6. Arbres parfait (a), et non parfaits (b) et (c).

• Un arbre binaire ***localement complet*** est un arbre binaire non vide qui n'a pas de point simple : tous les nœuds qui ne sont pas des feuilles ont deux fils.

Par exemple, les arbres binaires des figures 1 et 3 sont localement complets, mais celui de la figure 2 ne l'est pas.

On peut aussi donner une définition récursive des arbres binaires localement complets analogue à la définition récursive des arbres binaires :

$$\mathbb{BC} =< o,\ \emptyset,\ \emptyset > + < o,\ \mathbb{BC},\ \mathbb{BC} >$$

où $\mathbb{BC}$ représente l'ensemble des arbres binaires localement complets, et o désigne un nœud.

• Un ***peigne gauche*** (respectivement ***un peigne droit***) est un arbre binaire localement complet dans lequel tout fils droit (respectivement gauche) est une feuille (voir figure 7). Si $\mathbb{PG}$ représente l'ensemble des peignes gauches, on a donc

$$\mathbb{PG} =< o,\ \emptyset,\ \emptyset > + < o,\ \mathbb{PG},\ < o,\ \emptyset,\ \emptyset >> .$$

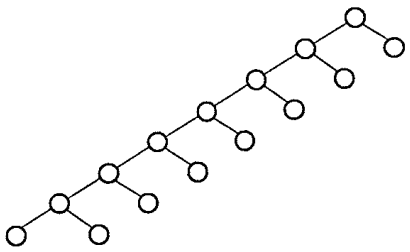

Figure 7. Arbre peigne gauche.

1.1.4. Occurrence et numérotation hiérarchique

Une façon classique de désigner un nœud dans un arbre est de lui associer un mot formé de symboles '0' et '1', qui décrit le chemin de la racine de l'arbre à ce nœud; ce mot est appelé ***occurrence*** du nœud dans l'arbre. Par définition l'occurrence de la racine d'un arbre est le mot vide ε et si un nœud de l'arbre a pour occurrence μ, son fils gauche a pour occurrence $\mu 0$, et son fils droit a pour occurrence $\mu 1$.

L'intérêt de cette notation est de permettre le codage d'un arbre par un ensemble de mots. Par exemple, l'arbre de la figure 4 est représenté par l'ensemble :

$$\{\varepsilon,\ 0,\ 1,\ 00,\ 10,\ 11,\ 001,\ 110,\ 0010,\ 0011\}$$

Il est facile de déterminer à partir de cet ensemble de mots les caractéristiques de l'arbre qu'il représente (cf. exercices).

De plus, ce codage des nœuds est lié à la numérotation en ***ordre hiérarchique***. Dans la numérotation en ordre hiérarchique des nœuds d'un arbre complet, on numérote en ordre croissant à partir de 1 tous les nœuds à partir de la racine, niveau par niveau, et de gauche à droite sur chaque niveau (voir figure 8). Ainsi un nœud numéroté i dans la numérotation en ordre hiérarchique a son fils gauche numéroté par $2i$, et son fils droit par $2i + 1$. Il en résulte que si un nœud d'un arbre complet a pour occurrence μ et pour numérotation en ordre hiérarchique i, on a la relation

$i = 2^{\lfloor \log_2 i \rfloor} + m$, où m est l'entier représenté par μ.

La preuve de ces propriétés est reportée en exercice.

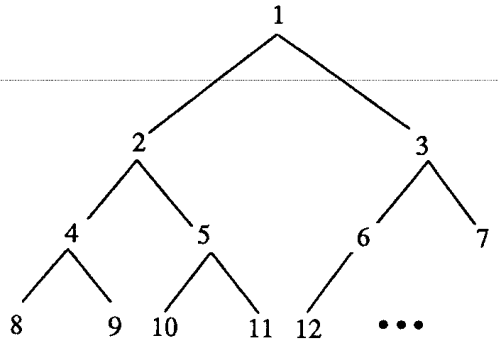

Figure 8. Ordre hiérarchique.

1.2. Propriétés fondamentales

La profondeur et la longueur de cheminement sont des quantités importantes pour l'analyse de la complexité d'un certain nombre d'algorithmes : on verra par la suite que la complexité, en temps ou en place, des algorithmes manipulant des arbres, s'exprime souvent en fonction de ces deux mesures. Il est donc intéressant de montrer les relations qui existent entre le nombre total de nœuds d'un arbre et son nombre de feuilles, sa hauteur ou sa longueur de cheminement. On donne aussi dans ce paragraphe des résultats de dénombrement d'arbres selon leur taille, qui sont utiles pour l'analyse de la complexité en moyenne.

1.2.1. Encadrements de la hauteur et de la longueur de cheminement

Lemme 1 : Pour un arbre binaire ayant n nœuds au total et de hauteur h , on a : $\lfloor \log_2 n \rfloor \leq h \leq n - 1$.

Preuve : Pour une hauteur h donnée, les arbres ayant le plus petit nombre de nœuds sont les arbres dégénérés (voir figure 5.a) ; et si B est un arbre dégénéré de hauteur h, alors sa taille est $n = h + 1$, d'où la seconde inégalité.

D'autre part, l'arbre de hauteur h ayant le plus grand nombre de nœuds est l'arbre complet (voir figure 5.b). Or, un arbre complet de hauteur h a pour taille $2^{h+1} - 1$.

Donc pour tout arbre binaire on a $n < 2^{h+1}$, ce qui implique $\log_2 n < h + 1$, d'où $\lfloor \log_2 n \rfloor \leq h$. □

Remarque : Un arbre binaire parfait avec n nœuds a pour hauteur $\lfloor \log_2 n \rfloor$. Cela provient de ce qu'un arbre binaire parfait de hauteur h contient entre 2^h nœuds et $2^{h+1} - 1$ nœuds.

Corollaire 1 : Tout arbre binaire non vide B ayant f feuilles a une hauteur $h(B)$ supérieure ou égale à $\lceil \log_2 f \rceil$.

Preuve : Montrons tout d'abord par récurrence que pour tout arbre binaire ayant n nœuds au total, dont f feuilles, on a $f \leq \frac{n+1}{2}$:

– la propriété est vraie pour l'arbre réduit à une feuille,
– supposons-la vraie pour tous les arbres binaires ayant moins de n nœuds.

Soit $B =< o,\ B1,\ B2 >$ un arbre ayant n nœuds et f feuilles. Notons n_1 et n_2 les nombres de nœuds de B_1 et de B_2, et f_1 et f_2 leurs nombres de feuilles respectifs; on a $n = n_1 + n_2 + 1$, et $f = f_1 + f_2$.

Par hypothèse de récurrence, on a $f_1 \leq \frac{n_1+1}{2}$ et $f_2 \leq \frac{n_2+1}{2}$, donc le nombre de feuilles de B est $f = f_1 + f_2 \leq \frac{n_1+n_2+2}{2} = \frac{n+1}{2}$, ce qui achève la preuve par récurrence.

Maintenant comme la fonction log est croissante sur $\mathbb{R}^+$, on a $\log_2 f \leq \log_2\left(\frac{n+1}{2}\right)$, soit $1 + \log_2 f \leq \log_2(n+1)$.

En prenant la partie entière supérieure on obtient $1 + \lceil \log_2 f \rceil \leq \lceil \log_2(n+1) \rceil$.
Il en résulte que $\lceil \log_2 f \rceil \leq \lfloor \log_2 n \rfloor$ (car $\lceil \log_2(n+1) \rceil = 1 + \lfloor \log_2 n \rfloor$).
En utilisant le lemme 1, on conclut donc que $\lceil \log_2 f \rceil \leq h(B)$. □

Lemme 2 : Soit B un arbre binaire ayant n nœuds au total, on a l'encadrement suivant pour la longueur de cheminement de B :

$$\sum_{1 \leq k \leq n} \lfloor \log_2 k \rfloor \leq LC(B) \leq \frac{n(n-1)}{2}$$

Preuve : Si l'arbre B est dégénéré, on a $LC(B) = 0 + 1 + 2 + ... + n - 1 = \frac{n(n-1)}{2}$.

A l'opposé, les arbres binaires qui minimisent la longueur de cheminement sont ceux qui ont toutes leurs feuilles situées sur deux niveaux au plus (voir exercices). On va supposer ici que les feuilles du dernier niveau sont le plus à gauche possible, et calculer la longueur de cheminement des arbres parfaits.

Soit B l'arbre parfait de taille n. Si les nœuds de B sont numérotés de 1 à n en ordre hiérarchique, alors le nœud numéroté k est à la profondeur $\lfloor \log_2 k \rfloor$; on montre cette propriété par récurrence sur la profondeur des nœuds d'un arbre parfait : le nœud numéroté 1 est à profondeur $0 = \lfloor \log_2 1 \rfloor$; supposons que les nœuds à profondeur k sont numérotés de 2^k à $2^{k+1} - 1$, et considérons le niveau

$k+1$: le nœud le plus à gauche a pour numéro $(2^{k+1}-1)+1 = 2^{k+1}$; de plus il y a 2 fois plus de nœuds qu'au niveau précédent; à profondeur $k+1$ il y a donc $2 \cdot 2^k$ nœuds, dont les numéros vont de 2^{k+1} à $2^{k+1}+2^{k+1}-1 = 2^{k+2}-1$.

Il en résulte que $LC(B) = \sum_{1 \leq k \leq n} \lfloor \log_2 k \rfloor$. □

Corollaire 2 : La longueur de cheminement d'un arbre binaire ayant n nœuds au total, est au minimum de l'ordre de $n \log_2 n$ et au maximum de l'ordre de n^2.

La preuve de ce corollaire est proposée en exercice. □

Il sera aussi utile pour la suite de connaître un minorant de la longueur de cheminement externe, et donc de la profondeur moyenne des feuilles, d'un arbre binaire ayant f feuilles.

Lemme 3 : Tout arbre binaire B ayant f feuilles a une profondeur moyenne externe $PE(B)$ supérieure ou égale à $\log_2 f$.

Preuve : On raisonne par récurrence. La propriété est vraie pour l'arbre réduit à une feuille, qui a pour profondeur 0. Supposons-la vraie pour tous les arbres ayant strictement moins de k feuilles. Soit $B =< o,\ B1,\ B2 >$ un arbre ayant k feuilles. B est de l'une des trois formes de la figure 9.

• Si B est de la forme a) ou b), alors B' est un arbre avec k feuilles, et l'on cherche à prouver la propriété sur B', ce qui la prouvera *a fortiori* sur B.

• Si B est de la forme c), ses sous-arbres gauche et droit ont strictement moins de k feuilles donc ils vérifient le lemme : si k_1 (respectivement k_2) est le nombre de feuilles de B_1 (respectivement B_2) on a $PE(B_1) \geq \log_2 k_1$ et $PE(B_2) \geq \log_2 k_2$;
or $PE(B) = 1 + \frac{k_1}{k_1+k_2}\ PE(B_1) + \frac{k_2}{k_1+k_2}\ PE(B_2)$
puisque $LCE(B) = k_1 + LCE(B_1) + k_2 + LCE(B_2)$; d'où

$$PE(B) \geq 1 + \frac{k_1}{k_1+k_2} \log_2 k_1 + \frac{k_2}{k_1+k_2} \log_2 k_2$$

Puisque $k_1 + k_2 = k$, le terme de droite de l'inéquation s'écrit :

$$q(k_2) = 1 + \frac{k-k_2}{k} \log_2 (k-k_2) + \frac{k_2}{k} \log_2 k_2$$

Pour k fixé, il est facile de montrer que $q(k_2)$ est minimum lorsque $k_2 = k/2$ (c'est en cette valeur que la dérivée s'annule), et l'on a $q(k/2) = \log_2 k$. En conséquence $PE(B) \geq \log_2 k$. □

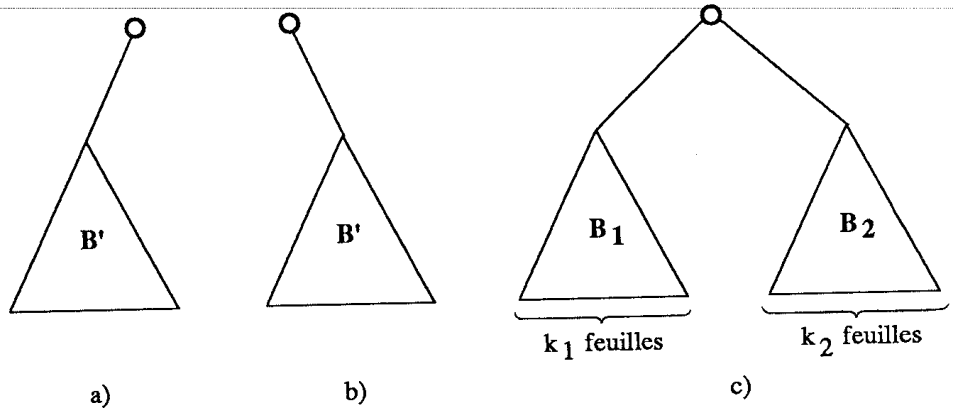

Figure 9

Remarque : Le lemme 3 et le corollaire 1 donnent des valeurs très proches pour les minorants de $h(B)$ et $PE(B)$, mais cela ne veut pas dire que les valeurs de $h(B)$ et $PE(B)$ sont très proches. Considérons l'arbre peigne B_0 de la figure 7. B_0 a 8 feuilles, donc $h(B_0) \geq 3$ d'après le corollaire1 et $PE(B_0) \geq 3$ d'après le lemme 3. Or, en fait, on calcule que $h(B_0) = 7$ et $PE(B_0) = 4.37$.

1.2.2. Propriétés des arbres binaires localement complets

Lemme 4 : Un arbre binaire localement complet ayant n nœuds internes a $(n+1)$ feuilles.

Preuve : La preuve se fait par récurrence sur le nombre de nœuds internes de l'arbre.

- La propriété est vraie pour l'arbre ayant 0 nœud interne et 1 feuille,
- Supposons-la vraie pour tous les arbres ayant moins de n nœuds internes, et soit $B =< o, B_1, B_2 >$ ayant n nœuds internes.

Notons n_1 (respectivement n_2) le nombre de nœuds internes de B_1 (respectivement B_2); on a : $n = n_1 + n_2 + 1$.

Par hypothèse de récurrence, B_1 (respectivement B_2) a $n_1 + 1$ (respectivement $n_2 + 1$) feuilles. Or les feuilles de B sont feuilles de B_1 ou de B_2, donc le nombre de feuilles de B est $(n_1 + 1) + (n_2 + 1) = n + 1$. □

Pour un arbre binaire localement complet, les longueurs de cheminement interne et externe vérifient la relation suivante.

Lemme 5 : Soit B un arbre binaire localement complet ayant n nœuds internes :

$$LCE(B) = LCI(B) + 2n.$$

La preuve est laissée en exercice. □

Par exemple, pour l'arbre binaire localement complet B de la figure 3, on a $n = 6$, $LCI(B) = 9$ et $LCE(B) = 21$.

On précise à présent le rapport entre les arbres binaires et les arbres binaires localement complets.

Définition : On appelle ***complétion locale d'un arbre binaire B*** l'opération qui consiste à compléter l'arbre B en un arbre BC, en rajoutant des feuilles de telle sorte que chaque nœud de B, interne ou externe, ait deux fils dans BC.

Par exemple sur la figure 10, l'arbre BC est obtenu par complétion locale de l'arbre B : on a noté par des carrés les feuilles de BC, qui sont les nœuds ajoutés par l'opération de complétion.

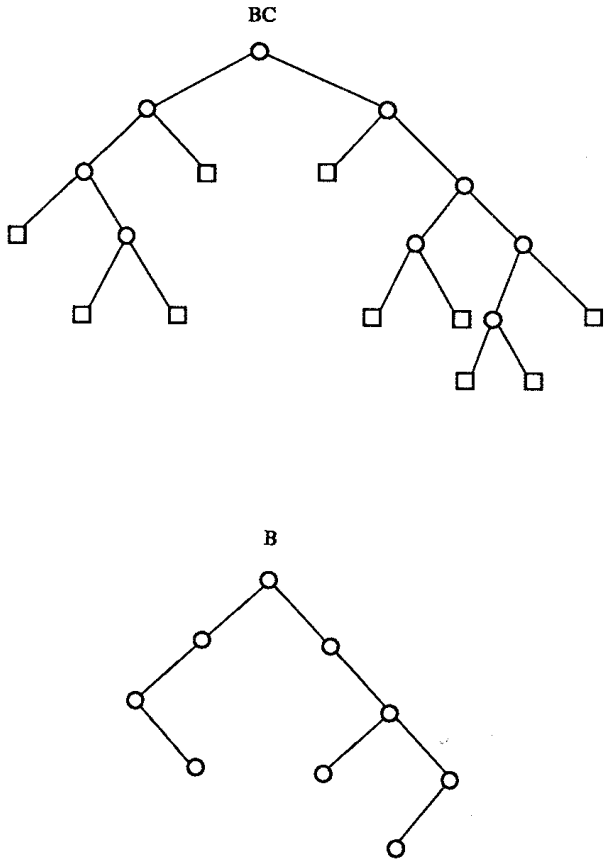

Figure 10. Complétion locale d'un arbre binaire.

Proposition 1 : Il existe une bijection entre l'ensemble $\mathbb{B}_n$ des arbres binaires ayant n nœuds, et l'ensemble $\mathbb{BC}_n$ des arbres binaires localement complets ayant $2n+1$ nœuds.

Preuve : D'après le lemme 4, si B est un arbre binaire localement complet ayant $2n+1$ nœuds, l'arbre B' obtenu en supprimant toutes les feuilles de B est un arbre binaire ayant n nœuds. Cette application est bijective : son inverse est l'opération de complétion locale. □

1.2.3. Dénombrement des arbres binaires

Pour faire une analyse en moyenne d'algorithmes opérant sur des arbres binaires, il est nécessaire de connaître le nombre b_n d'arbres binaires de taille donnée n. La figure 11 énumère les arbres binaires de taille $n = 0, 1, 2, 3$.

Proposition 2 : Le nombre d'arbres binaires de taille n est $b_n = \frac{1}{n+1}\binom{2n}{n}$.

La *preuve* de cette proposition est donnée dans le paragraphe sur les séries génératrices de l'annexe. □

Le nombre binômial, qui représente le nombre de façons de choisir n éléments parmi $2n$, a été noté ici $\binom{2n}{n}$; il est aussi quelquefois noté C_{2n}^{n}.

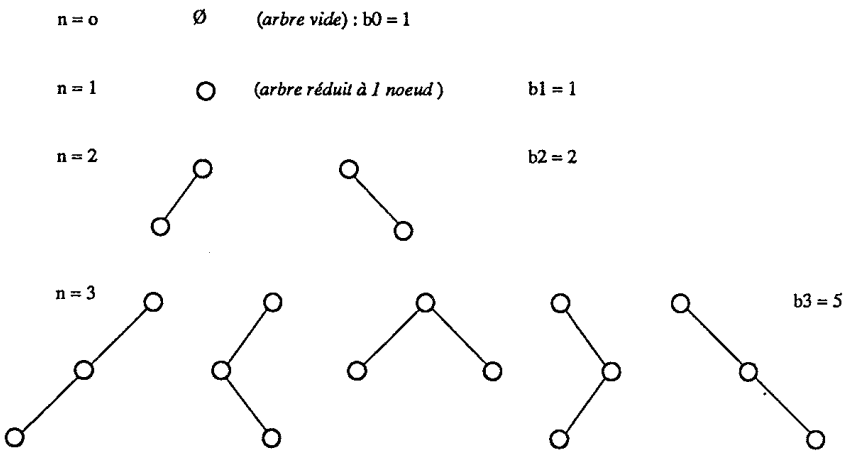

Figure 11. Enumération des arbres binaires de taille $0, 1, 2, 3$.

Comme conséquence des propositions 1 et 2, on a le corollaire suivant.

Corollaire 3 : Le nombre bc_n d'arbres binaires localement complets ayant $(2n+1)$ nœuds est $bc_n = \frac{1}{n+1}\binom{2n}{n}$.

La figure 12 montre les 5 arbres binaires localement complets avec 7 nœuds :

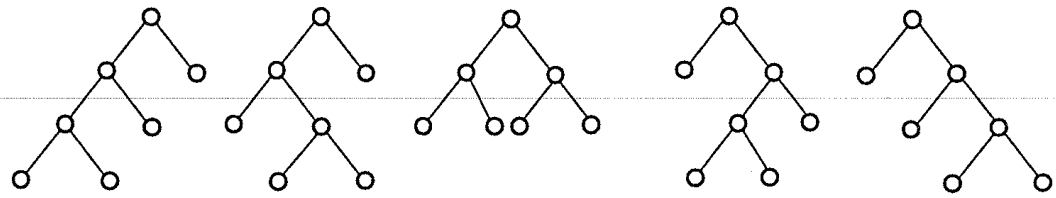

Figure 12. Arbres binaires localement complets avec 7 nœuds.

1.3. Représentation des arbres binaires

On décrit ici quelques façons d'implémenter les arbres binaires, qui seront utilisées par la suite. Il y a bien d'autres représentations possibles; dans une application donnée, c'est l'étude des opérations que l'on veut réaliser sur les arbres qui permet de déterminer la représentation la mieux adaptée au problème.

La représentation la plus naturelle reproduit la définition récursive des arbres binaires. Elle peut être réalisée en allouant la mémoire soit de façon chaînée soit de façon contiguë.

1.3.1. Utilisation de pointeurs

A chaque nœud on associe deux pointeurs, l'un vers le sous-arbre gauche, l'autre vers le sous-arbre droit, et l'arbre est déterminé par l'adresse de sa racine. Lorsque l'arbre est étiqueté, on représente dans un champ supplémentaire l'information contenue dans le nœud. Avec cette représentation, un arbre étiqueté comme celui de la figure 3 peut être défini par le type Pascal suivant :

```
type ARBRE = ↑ Nœud;
     Nœud = record val : Elément;
                   g, d : ARBRE
            end;
```

La figure 13 montre un arbre binaire A de type ARBRE. L'arbre vide ($A = nil$) est représenté par une case barrée.

Dans une telle représentation, les opérations du type abstrait Arbre donné au paragraphe 1.1 se traduisent comme suit : $A = nil$ correspond à $A =$ *arbre-vide*, $A\uparrow.val$ correspond à *contenu*(*racine*(A)), $A\uparrow.g$ correspond à $g(A)$, et $A\uparrow.d$ correspond à $d(A)$.

1.3.2. Utilisation de tableaux

Dans le cas où l'on n'a pas la possibilité d'allouer dynamiquement la mémoire, on peut simuler la représentation précédente à l'aide de tableaux : à chaque nœud de

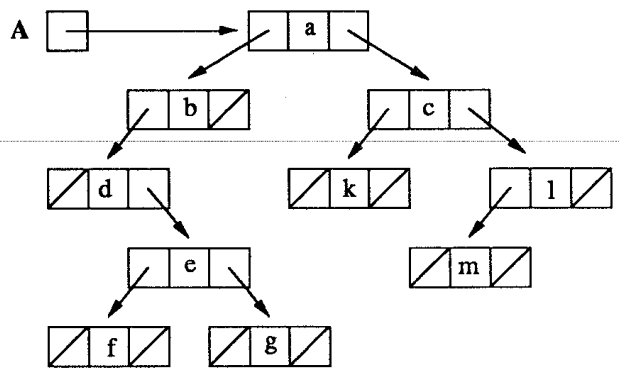

Figure 13. Représentation d'un arbre binaire à l'aide de pointeurs.

l'arbre on associe un indice dans un tableau à deux champs G et D. Le champ G (respectivement D) contient l'indice associé à la racine du sous-arbre gauche (respectivement droit). Il faut une convention pour l'indice associé à l'arbre vide. Il faut aussi connaître l'indice dans le tableau de la racine de l'arbre. De plus, lorsque l'arbre est étiqueté, on représente dans un champ supplémentaire V l'information contenue dans le nœud.

Avec cette représentation, un arbre étiqueté, contenant moins de N nœuds, peut être défini par le type Pascal ARBTAB, qui utilise le type TAB :

```
type    TAB = array [1..N] of record V : Elément;
                                     G : 0..N;
                                     D : 0..N
                              end;
      ARBTAB = record rac : 0..N;
                      tab : TAB
               end;
```

Dans ce cas, l'arbre de la figure 13 peut être représenté par la variable A de type ARBTAB, dont le champ *rac* contient 3, et le champ *tab* contient le tableau de la figure 14 ($N = 13$ est la valeur maximale de la taille des arbres qui peuvent être rangés dans ce tableau).

Voyons comment se traduisent les opérations du type abstrait Arbre sur une variable A de type ARBTAB. Par convention si $A.rac = 0$, on considère que A représente l'arbre vide. Sinon soit $r > 0$ tel que $r = A.rac$, alors $A.tab[r]$ correspond à *racine*(A), $A.tab[r].V$ est le contenu de cette racine, et $A.tab[r].G$ et $A.tab[r].D$ sont les indices de $g(A)$ et $d(A)$.

Notons que dans la représentation indexée, l'indice associé à chaque nœud de l'arbre est arbitraire; on peut donc ajouter ou enlever des nœuds de l'arbre sans difficulté, à condition de «chaîner» entre elles les cases libres de façon analogue à ce qui

	V	G	D
1			
2	d	0	10
3	a	5	6
4	g	0	0
5	b	2	0
6	c	13	11
7			
8	f	0	0
9	m	0	0
10	e	8	4
11	l	9	0
12			
13	k	0	0

Figure 14. Représentation indexée d'un arbre binaire.

a été fait au chapitre 5 pour les listes; on conserve ainsi, dans les limites de la place réservée pour le tableau, l'un des avantages de l'allocation chaînée. Avec des déclarations de types légèrement différentes, on pourrait représenter plusieurs arbres dans le même tableau, de façon analogue à ce qui a été fait pour les listes.

1.3.3. Cas particulier : les arbres binaires parfaits

Les arbres parfaits ont une représentation séquentielle très compacte qui repose sur la numérotation en ordre hiérarchique des nœuds d'un arbre dans un parcours par niveaux (figure 8).

L'utilisation de la numérotation en ordre hiérarchique permet de coder les arbres parfaits de taille n dans un tableau de n cases, comme le montre la figure 15. Si un nœud est numéroté par i dans la numérotation hiérarchique, alors son fils gauche est numéroté par $2i$, et son fils droit par $2i+1$. Donc dans cette représentation le passage d'un nœud à un autre se traduit par un simple calcul d'indice dans le tableau, calcul qu'il est bon de mémoriser dans le schéma suivant :

$2 \leq i \leq n \Rightarrow$ le père du nœud d'indice i est à l'indice i **div** 2
$1 \leq i \leq n$ **div** $2 \Rightarrow$ le fils gauche du nœud d'indice i est en $2i$
le fils droit du nœud d'indice i est en $2i+1$

Notons que cette représentation peut être utilisée pour des arbres binaires quelconques, en laissant des cases vides dans le tableau, qui marquent la place des nœuds

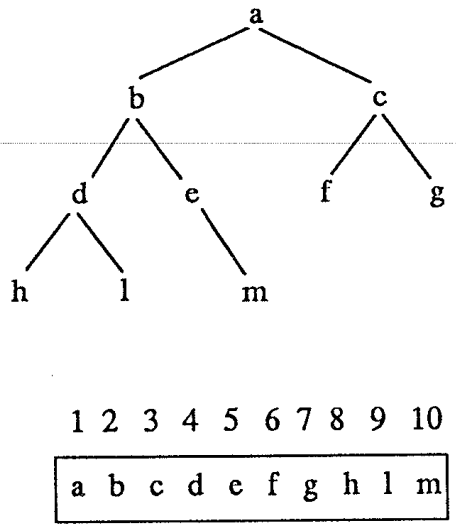

Figure 15. Représentation d'un arbre parfait dans un tableau.

possibles non présents. Mais elle perd alors son avantage de compacité : la représentation d'un arbre à n nœuds peut nécessiter jusqu'à $2^n - 1$ cases, comme le montre l'exemple de la figure 16.

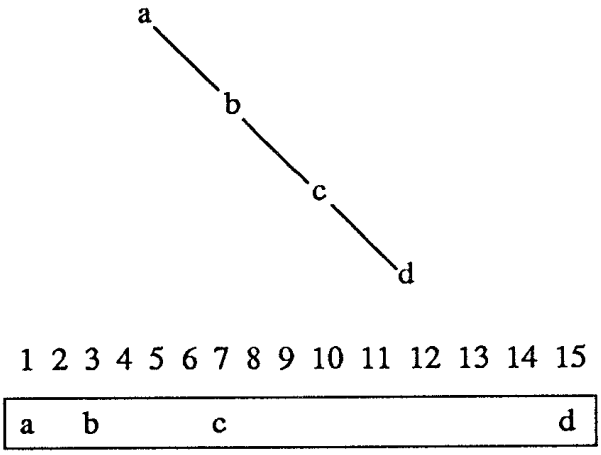

Figure 16. Représentation d'une branche droite par un tableau.

1.4. Parcours d'un arbre binaire

Une des opérations les plus fréquentes mise en oeuvre par les algorithmes qui manipulent des arbres consiste à examiner systématiquement, dans un certain ordre, chacun des nœuds de l'arbre pour y effectuer un même traitement; cette opération est appelée ***parcours*** ou ***traversée*** de l'arbre. On étudie ici un algorithme général de parcours d'un arbre binaire, que l'on appelle parcours en profondeur à main gauche

– sous forme récursive et sous forme itérative – et trois ordres classiques induits sur les nœuds.

Le ***parcours en profondeur à main gauche*** consiste à tourner autour de l'arbre en suivant le chemin indiqué sur la figure 17. Ce chemin part à gauche de la racine, et va toujours le plus à gauche possible en suivant l'arbre (notons que l'on a fait figurer les sous-arbres vides de l'arbre binaire).

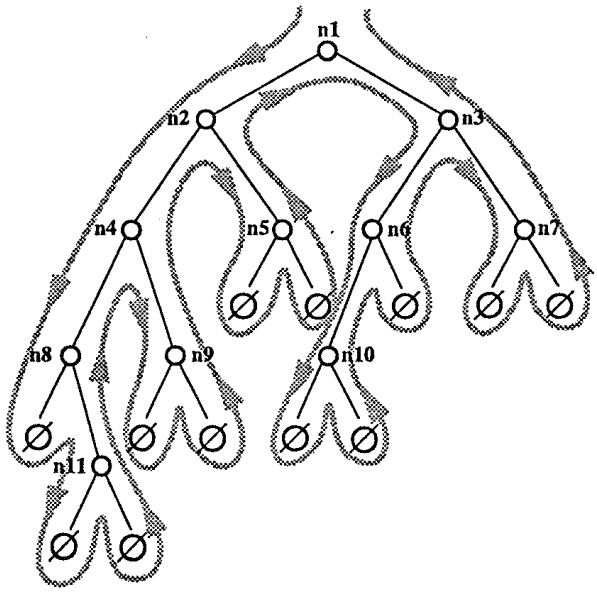

Figure 17. Parcours en profondeur à main gauche.

Dans ce parcours, chaque nœud de l'arbre est rencontré trois fois : d'abord à la descente (figure 18.a), puis en montée gauche, après son sous-arbre gauche (figure 18.b), et enfin en montée droite, après son sous-arbre droit (figure 18.c).

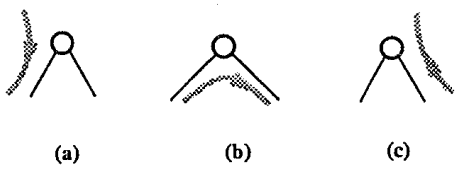

Figure 18. Rencontres des nœuds lors d'un parcours.

1.4.1. Algorithme récursif de parcours

Lors du parcours en profondeur à main gauche de l'arbre A, on peut faire correspondre à chacune des rencontres d'un nœud un traitement particulier pour ce nœud. Appelons TRAITEMENT1 (resp. TRAITEMENT2 ET TRAITEMENT3) la suite d'actions exécutées lorsqu'un nœud est rencontré pour la première (resp. deuxième et troisième) fois; de plus, envisageons pour les arbres vides un traitement

spécial : TERMINAISON. Le parcours de l'arbre est alors décrit récursivement par la procédure *Parcours* ci-dessous. Remarquons ici que la complexité, comptée en nombre de traitements de nœuds, de l'algorithme de parcours, sur un arbre comportant n nœuds , est clairement en $\Theta(n)$.

```
procedure Parcours(A : Arbre);
begin
    if A = arbre-vide then TERMINAISON
    else     begin        TRAITEMENT1;
                          Parcours(g(A));
                          TRAITEMENT2;
                          Parcours(d(A));
                          TRAITEMENT3
             end
end Parcours;
```

Le parcours en profondeur à main gauche contient comme cas particuliers trois ordres classiques d'exploration d'arbres :

1) ***ordre préfixe*** ou ***préordre*** : si TRAITEMENT2 et TRAITEMENT3 n'existent pas, alors TRAITEMENT1 est appliqué aux nœuds de l'arbre en ordre préfixe; sur l'exemple de la figure 17 cela correspond à l'ordre de traitement n_1, n_2, n_4, n_8, n_{11}, n_9, n_5, n_3, n_6, n_{10}, n_7;

2) ***ordre infixe*** ou ***symétrique*** : c'est l'ordre obtenu lorsque seul TRAITEMENT2 (et TERMINAISON) sont appliqués; sur l'exemple cela correspond à l'ordre de traitement n_8, n_{11}, n_4, n_9, n_2, n_5, n_1, n_{10}, n_6, n_3, n_7;

3) ***ordre suffixe*** ou ***postfixe*** : c'est l'ordre obtenu lorsque seul TRAITEMENT3 (et TERMINAISON) sont appliqués; sur l'exemple cela correspond à l'ordre de traitement n_{11}, n_8, n_9, n_4, n_5, n_2, n_{10}, n_6, n_7, n_3, n_1.

Remarque : On ne peut pas obtenir ainsi l'ordre hiérarchique (figure 8), car il correspond à un *parcours par niveaux* de l'arbre et non à un parcours en profondeur.

1.4.2. Arbre étiqueté représentant une expression arithmétique

Une expression arithmétique peut être représentée par un arbre binaire dont les feuilles contiennent des symboles de variables (x, y, z...) ou de constantes (2, 3...) et les nœuds internes contiennent des symboles d'opérateurs (cf. figure 19). Si l'on ne considère que des opérateurs binaires, comme +, − , *, /, l'arbre binaire est localement complet.

Pour un tel arbre, les suites des éléments en ordre préfixe, suffixe et symétrique, correspondent aux différentes représentations habituelles d'une expression arithmétique :

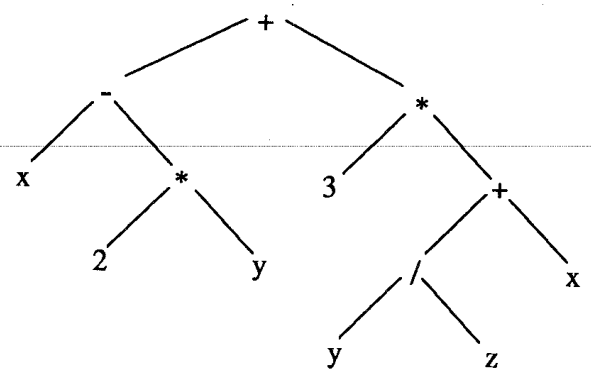

Figure 19. Arbre binaire représentant une expression arithmétique.

- en ordre préfixe, on obtient l'expression sous forme polonaise préfixée : chaque opérateur précède ses deux opérandes; ce qui donne sur l'exemple de la figure 19 : $+ - x * 2y * 3 + /yzx$.
- en ordre suffixe, on obtient l'expression sous forme polonaise postfixée : chaque opérateur est précédé par ses deux opérandes; ce qui donne pour l'exemple : $x2y * -3yz/x + *+$

Remarquons que pour les deux formes polonaises, la notation est non ambiguë, et il est donc inutile de mettre des parenthèses autour des sous-expressions.

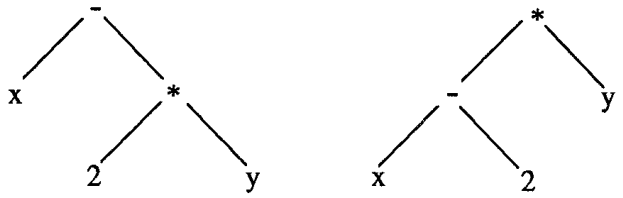

Figure 20. Ambiguïté de la notation symétrique.

Par contre la lecture des éléments de l'arbre en ordre symétrique produit la notation symétrique habituelle, mais sans parenthèses, ce qui la rend ambiguë : par exemple, l'expression $x - 2 * y$ représente les deux arbres différents de la figure 20.

L'ambiguïté peut être levée en introduisant des parenthèses. Les deux arbres de la figure 20 sont respectivement représentés par les expressions bien parenthésées $(x - (2 * y))$ et $((x - 2) * y)$. L'algorithme ci-dessous écrit l'expression symétrique bien parenthésée correspondant à un arbre binaire non vide A. On utilise l'opération *feuille* de profil

feuille : Arbre $\rightarrow$ Booléen,

et qui satisfait la propriété

feuille(A) = vrai **ssi** $A \neq$ *arbre-vide* & $g(A) =$*arbre-vide* & $d(A) =$ *arbre-vide*.

```
procedure Sym (A : Arbre);
{Cette procédure écrit l'expression symétrique bien parenthésée correspondant
à l'arbre binaire A supposé non vide}
begin
    if feuille(A) then write(contenu(racine(A)))
        else begin write('(');
                   Sym(g(A));
                   write(contenu(racine(A)));
                   Sym(d(A));
                   write (')')
             end
end Sym;
```

1.4.3. Version itérative du parcours d'arbres binaires

La procédure *Parcours* est une version récursive du parcours en profondeur à main gauche d'un arbre binaire. Dans ce parcours chaque nœud est rencontré trois fois : une première fois à la descente, et l'on effectue le TRAITEMENT1 (en abrégé T_1), une deuxième fois en remontée par la gauche et l'on effectue le TRAITEMENT2 (T_2) et une troisième fois en remontée par la droite et l'on effectue le TRAITEMENT3 (T_3). Par exemple sur l'arbre de la figure 17, si l'on note (T_i, n_k) l'action d'effectuer T_i sur le nœud n_k, le parcours général donne la suite :
(T_1, n_1), (T_1, n_2), (T_1, n_4), (T_1, n_8), (T_2, n_8), (T_1, n_{11}), (T_2, n_{11}), (T_3, n_{11}), (T_3, n_8), (T_2, n_4), (T_1, n_9), (T_2, n_9), (T_3, n_9), (T_3, n_4), (T_2, n_2), (T_1, n_5), (T_2, n_5), (T_3, n_5), (T_3, n_2), (T_2, n_1), (T_1, n_3), (T_1, n_6), (T_1, n_{10}), (T_2, n_{10}), (T_3, n_{10}), (T_2, n_6), (T_3, n_6), (T_2, n_3), (T_1, n_7), (T_2, n_7), (T_3, n_7), (T_3, n_3), (T_3, n_1).

L'examen du parcours général suggère l'algorithme itératif suivant : descendre la branche gauche de l'arbre en effectuant T_1, et en empilant les nœuds rencontrés; c'est avec ces nœuds qu'il faudra effectuer T_2 lors de la remontée par la gauche; il est donc nécessaire de les conserver. De plus l'ordre de rencontre des nœuds pour T_2 est inverse de l'ordre de rencontre pour T_1, d'où la structure de pile pour conserver l'information; ensuite lorsqu'un nœud est rencontré en remontée gauche il faut effectuer T_2, mémoriser le nœud dans une pile, puis il faut travailler sur tout son sous-arbre droit, et enfin traiter le nœud lui-même en remontée droite par T_3.

Pour éviter d'utiliser deux piles, on introduit une marque N qui vaut 1 lorsque le nœud est rencontré en descente et en remontée gauche, et 2 lorsqu'il est rencontré en remontée droite. On utilise dans l'algorithme le type Pile et les opérations *empiler*, *dépiler* et *est-vide* définies sur ce type au chapitre 5. La pile est formée d'éléments qui sont des couples (Arbre, marque). Le parcours itératif d'arbres binaires se décrit comme suit.

```
procedure Parcoursiter (A : Arbre);
var P : Pile; N : integer;
{N est une marque, positionnée au moment de la descente, qui indique au moment de la remontée si on remonte du sous-arbre gauche (N = 1) ou droit (N = 2)}
begin
    N = 1; P :=pile-vide;
    repeat
        if N= 1 then begin
            while A <> arbre-vide do begin
              TRAITEMENT1;
              P := empiler(P, (A, 1));
              A := g(A) {descente gauche}
              end;
            TERMINAISON {traitement de l'arbre vide}
        end;
        if not est-vide(P) then begin
              (A, N) := sommet(P); P := dépiler(P);
              if N = 1 then begin
              TRAITEMENT2; {remontée de la gauche}
              P := empiler(P, (A, 2));
              A := d(A) {descente droite}
              end
              else TRAITEMENT3 {remontée de la droite}
        end
    until est-vide(P)
end Parcoursiter;
```

Remarque : La méthode qui vient d'être présentée pour transformer le parcours récursif d'arbres binaires en parcours itératif peut être adaptée pour dérécursifier beaucoup d'autres algorithmes comportant deux appels récursifs; on en verra des exemples par la suite.

2. Arbres planaires généraux

On présente maintenant une structure arborescente plus large appelée ***arbre planaire général*** ou plus brièvement *arbre général* ou *arbre*, où le nombre de fils de chaque nœud n'est plus limité à deux. La figure 21 montre un exemple d'arbre planaire général.

2.1. Définitions et propriétés

Un ***arbre*** $A =< o,\ A_1, \ldots,\ A_p >$ est la donnée d'une racine et d'une liste finie, éventuellement vide (si $p = 0$), d'arbres disjoints. Une liste finie éventuellement

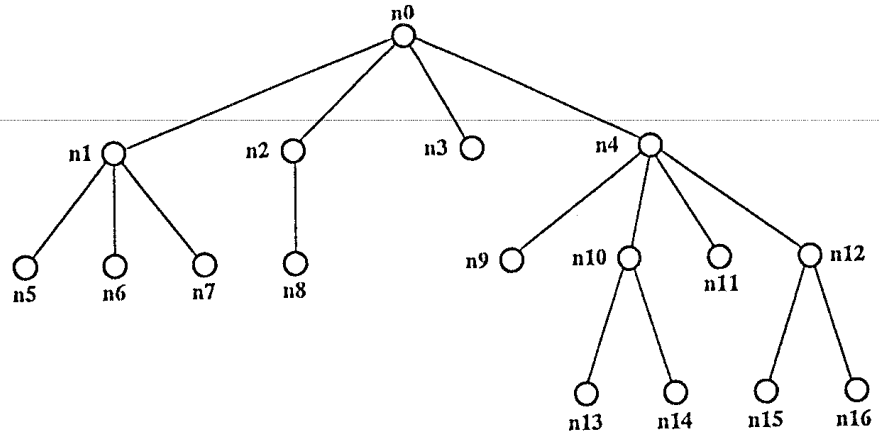

Figure 21. Arbre planaire général.

vide d'arbres disjoints est appelée une ***forêt***. Un arbre est donc obtenu en ajoutant un nœud racine à une forêt.

Ces définitions peuvent être exprimées par les équations :

$$\mathbb{A} = < o,\ \mathbb{F} >$$
$$\text{avec } \mathbb{F} = \emptyset + \mathbb{A} + < \mathbb{A},\ \mathbb{A} > + < \mathbb{A},\ \mathbb{A},\ \mathbb{A} > + \ldots$$

où $\mathbb{A}$ désigne l'ensemble des arbres planaires généraux, $\mathbb{F}$ l'ensemble des forêts et $\emptyset$ la forêt ne contenant aucun arbre.

La signature correspondant à ces définitions est la suivante :

sorte Arbregen, Forêt
utilise Nœud, Entier
opérations

cons	:	Nœud × Forêt → Arbregen
racine	:	Arbregen → Nœud
list-arbres	:	Arbregen → Forêt
forêt-vide	:	→ Forêt
ième	:	Forêt × Entier → Arbregen
nb-arbres	:	Forêt → Entier
insérer	:	Forêt × Entier × Arbregen → Forêt

Cette signature peut être complétée par d'autres opérations, par exemple *supprimer* un arbre d'une forêt, ou donner le *contenu* d'un nœud. Dans un but de simplification on se limite ici aux opérations de construction et d'accès pour les arbres et les forêts.

Etant donné des variables, o de sorte Nœud, F de sorte Forêt, A de sorte Arbregen et i de sorte Entier, on a la précondition et les axiomes suivants :

précondition

insérer(F, i, A) **est-défini-ssi** $1 \leq i \leq 1+$ *nb-arbres*(F)
{*l'opération insérer permet d'ajouter un arbre à une forêt*}

axiomes

racine(*cons*(o, F)) $= o$
list-arbres(*cons*(o, F)) $= F$
nb-arbres(*forêt-vide*) $= 0$
$1 \leq i \leq 1+$ *nb-arbres*(F) $\Rightarrow$
nb-arbres(*insérer*(F, i, A)) $=$ *nb-arbres*(F) $+ 1$
$1 \leq k < i \Rightarrow$ *ième*(*insérer*(F, i, A), k) $=$ *ième*(F, k)
$k = i \Rightarrow$ *ième*(*insérer*(F, i, A), k) $= A$
$i + 1 \leq k \leq 1 +$ *nb-arbres*(F) $\Rightarrow$ *ième*(*insérer*(F, i, A), k) $=$ *ième*($F, k-1$)

Remarque : Il est important de remarquer qu'il n'y a pas de notion gauche-droite dans les arbres : alors que $< o, < o, \emptyset, \emptyset >, \emptyset >$ et $< o, \emptyset, < o, \emptyset, \emptyset >>$ sont deux arbres binaires différents, il n'y a qu'un arbre planaire général avec deux nœuds l'arbre $< o, < o, \emptyset >>$. Une autre différence importante avec les arbres binaires, est qu'un arbre n'est jamais vide !

Le vocabulaire concernant les arbres est le même que celui présenté précédemment pour les arbres binaires à une différence près : il n'y a plus de notion gauche-droite pour les fils et pour les sous-arbres. Les fils (et les sous-arbres) d'un nœud sont ordonnés et l'on parle du premier fils, du deuxième fils, du troisième fils... et de même pour les sous-arbres. Par exemple sur la figure 21, n_9 est le premier fils de n_4, et n_{12} est le quatrième fils de n_4, n_8 est le premier fils (et fils unique) de n_2, n_{13} est le premier fils de n_{10} et n_{14} son deuxième fils.

2.2. Parcours des arbres généraux

Par extension du parcours des arbres binaires vu au paragraphe précédent, on peut définir un parcours en profondeur à main gauche des arbres généraux; dans ce parcours chaque nœud de l'arbre est rencontré une fois de plus que son nombre de fils, et l'on peut faire correspondre à chacun de ces passages un certain traitement du nœud rencontré : on note TRAITEMENT(i) (en abrégé T_i) la suite d'actions exécutées lors du $(i+1)^{ième}$ passage sur le nœud (figure 22); le premier traitement sur un nœud est noté TRAITEMENTPREF et le dernier traitement TRAITEMENTSUFF; on note TERM le traitement particulier des nœuds qui n'ont pas de fils.

La procédure *Parc* est une version récursive du parcours d'un arbre général A. Sa complexité, comptée en nombre de traitements de nœuds, est en $\Theta(n)$, si n est le nombre de nœuds de l'arbre A.

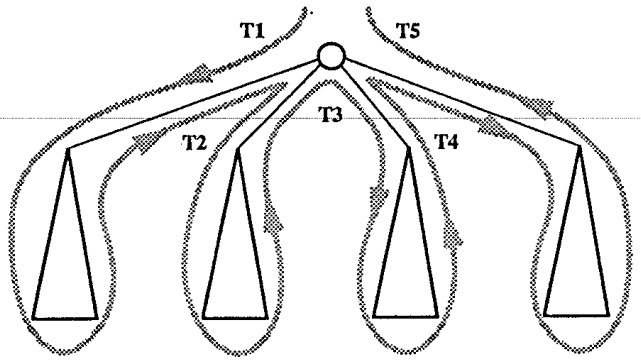

Figure 22. Parcours d'un arbre général.

L'opération *feuille*(A) teste si un arbre A est réduit à un seul nœud, c'est-à-dire si *list-arbres*(A) = *forêt-vide.*

```
procedure Parc(A : Arbregen);
var i, nb : integer;
begin
    nb := nb-arbres(list-arbres(A));
    if feuille(A) then TERM
    else begin
        TRAITEMENTPREF; {traitement avant de voir les fils}
        for i := 1 to nb - 1 do begin
            Parc(ième(list-arbres(A), i));
            TRAITEMENT(i)
        end;
        Parc(ième(list-arbres(A), nb));
        TRAITEMENTSUFF {traitement après avoir vu tous les fils}
    end
end Parc;
```

Deux ordres naturels d'exploration des arbres généraux sont inclus dans ce parcours.

1) L'***ordre préfixe***, où chaque nœud n'est pris en compte que lors du premier passage : cela revient à ne faire intervenir que TRAITEMENTPREF et TERM dans l'algorithme de parcours.

2) L'***ordre suffixe***, où chaque nœud n'est pris en compte que lors du dernier passage : dans l'algorithme de parcours cela revient à ne faire intervenir que TRAITEMENTSUFF et TERM .

Exemple : La figure 21 est la représentation graphique d'un arbre; on peut aussi représenter cet arbre linéairement à l'aide de parenthèses :

(a) $(n_o(n_1(n_5)(n_6)(n_7))(n_2(n_8))(n_3)(n_4(n_9)(n_{10}(n_{13})(n_{14}))(n_{11})(n_{12}(n_{15})(n_{16}))))$

Cette façon de noter doit être familière aux utilisateurs du langage LISP; ceux qui ont surtout l'habitude d'écrire les fonctions dans la notation mathématique usuelle préfèrent peut-être la représentation linéaire de l'arbre à l'aide de parenthèses et de virgules :

(b) $n_o(n_1(n_5,\ n_6,\ n_7),\ n_2(n_8),\ n_3,\ n_4(n_9,\ n_{10}(n_{13},\ n_{14}),\ n_{11},\ n_{12}(n_{15},\ n_{16})))$

Pour obtenir une représentation des arbres généraux sous la forme (a) il suffit que les traitements de la procédure de parcours *Parc* soient les suivants :

- TRAITEMENTPREF écrit une parenthèse ouvrante suivie du contenu de la racine de A;
- TRAITEMENTSUFF écrit une parenthèse fermante;
- le traitement, réservé aux feuilles, TERM écrit une parenthèse ouvrante, puis le nœud, puis une parenthèse fermante;
- les autres traitements ne font rien.

Pour obtenir une représentation sous la forme (b) il suffit que dans la procédure de parcours *Parc* on ait les traitements suivants :

- TRAITEMENTPREF écrit le contenu de la racine de A et une parenthèse ouvrante;
- les TRAITEMENT(i) écrivent une virgule;
- TRAITEMENTSUFF écrit une parenthèse fermante;
- le traitement réservé aux feuilles, TERM, écrit le contenu du nœud.

2.3. Représentation des arbres généraux

2.3.1. Représentations simples

Différentes façons d'implémenter un arbre sont envisageables.

On peut donner pour chaque nœud la liste de ses fils comme sur la figure 23, qui représente l'arbre de la figure 21. Cette représentation des arbres peut être utile, mais elle se prête mal à une gestion dynamique (cf. exercices).

On peut aussi décrire une représentation analogue à celle des arbres binaires : chaque nœud contient un pointeur vers chacun des sous-arbres (cf. figure 24), et éventuellement un champ pour stocker l'élément contenu dans le nœud. Sauf pour des cas particuliers d'arbres dont tous les nœuds ont à peu près le même nombre de fils, cette représentation est trop gourmande en place (c'est le nombre maximal de

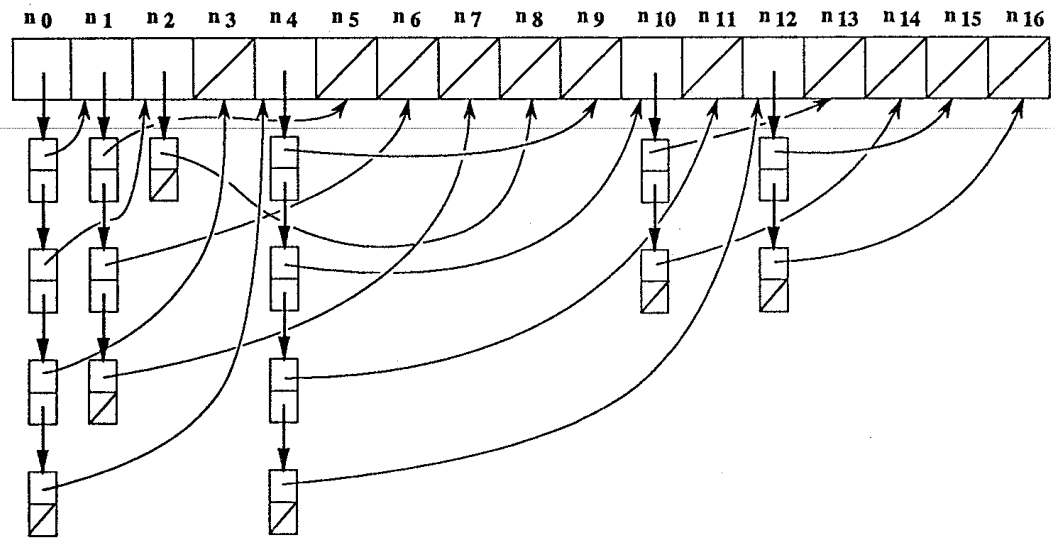

Figure 23. Représentation d'un arbre par liste de fils.

fils d'un nœud qui détermine le nombre de pointeurs dans tous les nœuds), et donc peu utilisable.

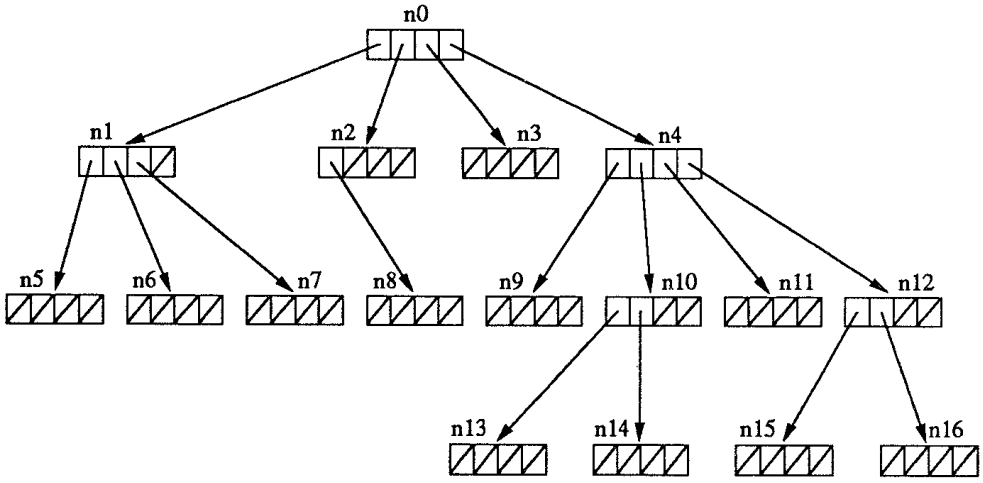

Figure 24. Représentation d'un arbre par des n-uplets de pointeurs.

La suite de ce paragraphe est consacrée à une représentation, très utilisée, des arbres généraux sous forme d'arbres binaires.

2.3.2. Représentation sous forme d'arbres binaires

On établit ici une correspondance entre les arbres planaires généraux et les arbres binaires. Rappelons qu'un arbre général est obtenu en ajoutant un nœud racine à

une forêt. De même qu'on a défini la taille d'un arbre binaire, on définit la ***taille d'un arbre général*** comme le nombre total de ses nœuds; la ***taille d'une forêt*** est le nombre total de nœuds de tous les arbres qui la composent. On a donc :

taille(forêt-vide) = 0
taille(insérer(F, i, A)) = *taille(F)* + *taille(A)*
taille(cons(o, F)) = 1 + *taille(F)*

Lemme 7 : Il existe une bijection entre les arbres généraux ayant $n + 1$ nœuds, et les forêts de taille n.

La *preuve* est évidente : on associe à l'arbre $A =< o, A_1, \ldots, A_p >$ la forêt $F =< A_1, \ldots, A_p >$; l'arbre A a un nœud de plus (sa racine) que la forêt. La figure 25 montre la forêt associée à l'arbre de la figure 21. □

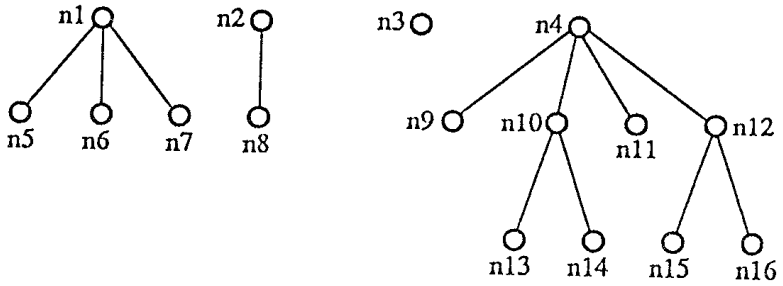

Figure 25. Forêt associée à l'arbre de la figure 21.

Montrons maintenant que l'on peut coder les forêts par des arbres binaires. La figure 26 montre l'arbre binaire associé à la forêt de la figure 25 dans la bijection dite ***bijection fils aîné-frère droit.***

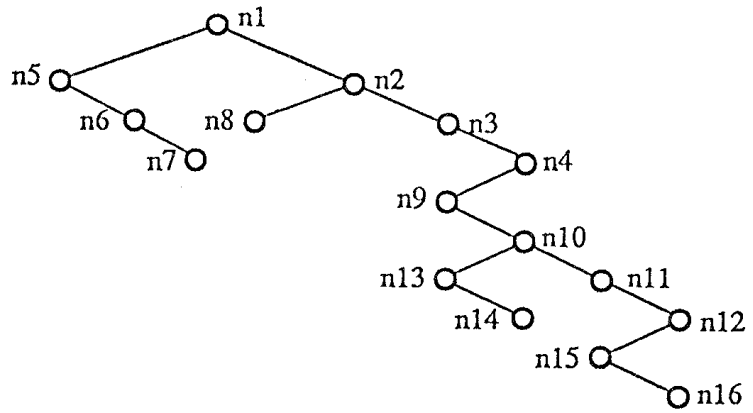

Figure 26. Arbre binaire associé à la forêt de la figure 25.

Proposition 3 : Il existe une bijection entre les forêts de taille n et les arbres binaires ayant n nœuds.

Preuve : Pour obtenir l'arbre binaire associé à une forêt, on construit pour chaque nœud un lien gauche vers son *premier fils* (fils aîné), et un lien droit vers son *frère* situé immédiatement à sa droite dans la forêt (on considère que les racines des différents arbres de la forêt sont des nœuds frères), comme le montre le schéma de la figure 27.

Inversement, pour revenir de l'arbre binaire vers la forêt qu'il représente, on effectue la transformation inverse de celle de la figure 27 : conserver les liens gauches; supprimer les liens droits, et construire des liens entre un nœud et chacun des nœuds du bord droit de son ancien sous-arbre gauche. □

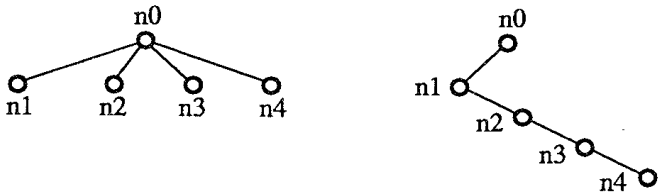

Figure 27. Bijection fils aîné-frère droit.

Les deux résultats précédents permettent maintenant :
- d'une part de dénombrer les arbres généraux (respectivement les forêts) selon leur taille,
- d'autre part de donner une bonne représentation des arbres (respectivement des forêts) en termes d'arbres binaires.

Proposition 4 : Le nombre a_{n+1} d'arbres de taille $n+1$ est :

$$a_{n+1} = \frac{1}{n+1}\binom{2n}{n}$$

La *preuve* est une conséquence immédiate du lemme 7 et des propositions 2 et 3 : il y a autant de forêts de taille n, que d'arbres binaires ayant n nœuds, et il y a autant d'arbres ayant $(n+1)$ nœuds que de forêts de taille n. □

La figure 28 montre les 5 arbres que l'on peut construire avec 4 nœuds.

Proposition 5 : Dans la bijection fils aîné-frère droit entre une forêt F et un arbre binaire B,
- le bord gauche de B est le même que le bord gauche du premier arbre de F,
- le bord droit de B représente la suite des racines des arbres de F,

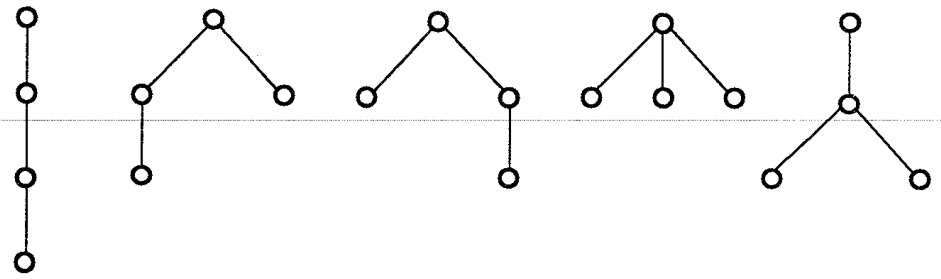

Figure 28. Les 5 arbres avec 4 nœuds.

- le parcours préfixe de B donne les nœuds dans le même ordre que le parcours préfixe de F,
- le parcours symétrique de B donne les nœuds dans le même ordre que le parcours suffixe de F.

Cette propriété, dont la preuve est laissée en exercice, montre que la représentation des arbres et des forêts par des arbres binaires est utilisable en pratique : les opérations sur les arbres ne sont en général pas difficiles à décrire sur les arbres binaires leur correspondant.