[SD 010] Arbres

Arbres[1]

Dans ce chapitre, on va s'intéresser à des structures à deux dimensions avec liens appelées ***arbres*** qui sont au coeur d'un grand nombre d'algorithmes. Nous étudierons les définitions premières et la terminologie associée aux arbres. Nous déduirons aussi certaines de leurs propriétés les plus importantes et nous examinerons les modes de représentation informatique de ces structures fondamentales sur lesquelles opèrent de nombreux algorithmes.

Les arbres sont fréquents dans notre vie quotidienne. Par exemple : Arbres généalogiques (une grande partie de la terminologie découle de cet usage), organisation de compétitions sportives, organigrammes d'entreprises, arbres syntaxiques de phrases du langage courant,...

VOCABULAIRE

Un ***arbre*** est une collection non vide de ***nœuds*** (ou ***sommets***) et ***d'arêtes*** assujettis à certaines conditions. Un nœud est un objet simple qui peut porter un nom ainsi qu'un certain nombre d'informations pertinentes. Une arête est un lien entre deux nœuds. Une ***branche*** (ou ***chemin***) de l'arbre est une suite de nœuds distincts dans laquelle deux nœuds successifs sont reliés par une arête. Un nœud de l'arbre est spécifiquement désigné comme étant la ***racine***. La propriété qui définit un arbre est qu'il existe exactement une branche entre la racine et chacun des autres nœuds de l'arbre. S'il existe plus d'une branche entre la racine et un **nœud,** ou pas de branche du tout, on n'est plus en présence d'un **arbre,** mais d'un ***graphe***..

Bien que la définition donnée précédemment n'implique aucune notion de "direction" sur les arêtes, il est habituel de se représenter les arêtes comme "s'éloignant toutes de la racine" (direction vers le bas sur la figure 1) ou au contraire "allant toutes vers la racine" (direction vers le haut sur la même figure), suivant les cas. il est aussi habituel de représenter les arbres avec la racine en haut (bien que cela paraisse un peu bizarre au début). On dit d'un nœud x qu'il est ***en dessous*** d'un autre nœud y (ou encore que y est ***au-dessus*** de x) s'il en est ainsi sur cette représentation graphique de l'arbre ; c'est à dire si la branche qui va de la racine au nœud x passe par le nœud y. Chaque nœud, à l'exception de la racine, possède ainsi juste au-dessus de lui un nœud appelé son ***père*** ou ***parent***. Les nœuds directement situés sous un nœud donné sont ses ***enfants*** ou ***fils***. on pousse parfois l'analogie généalogique en parlant de "grand-père", voire de "frères" d'un nœud. Sur la figure 1, T est le petit-fils de O et il a trois frères.

[1] texte établi d'après les livres suivants, en cherchant à corriger les erreurs ou imprécisions :
Robert Sedgewick - Algorithmes en langage C - InterÉditions 1991
Ellis Horowitz et Sartaj Sahni - Data Structures in Pascal - Computer Science Press 1987

Les nœuds sans descendance sont appelés ***feuilles*** ou nœuds ***terminaux***, tandis qu'un nœud avec descendance est dit ***non terminal***. Les nœuds non terminaux sont fréquemment différents des autres : par exemple, ils peuvent ne pas porter de nom ni d'information particulière. Dans de tels cas, on dit souvent que les nœuds terminaux sont ***externes*** et les nœuds non terminaux ***internes***.

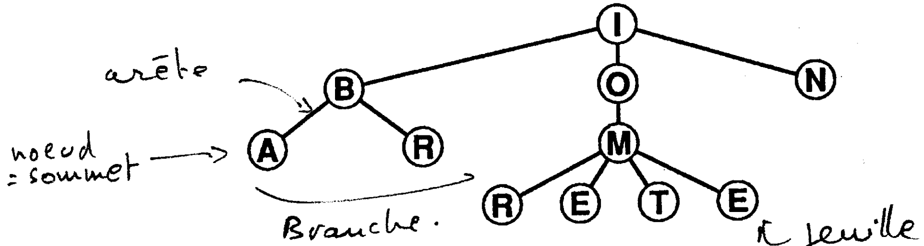

Figure 1. Un arbre type.

Tout nœud est la racine du sous-arbre constitué par sa descendance et lui-même. Sur l'arbre de la figure 1, on dénombre sept sous-arbres d'un nœud, un sous-arbre de trois nœuds, un sous-arbre de cinq nœuds et un sous-arbre de six nœuds. Un ensemble d'arbres est appelé une ***forêt*** : par exemple, si l'on supprime de l'arbre de la figure 1 la racine et les branches qui la relient au reste de l'arbre, on obtient une forêt constituée de trois arbres de racines **B,O** et **N.**

L'ordre dans lequel les enfants de chaque nœud sont rangés est parfois important parfois indifférent. Un arbre est dit ***ordonné*** si l'ordre des enfants de chacun de ses nœuds est spécifié.

Les nœuds d'un arbre se répartissent en ***niveaux*** : On donne par convention le niveau *1* à la racine, puis on incrémente le niveau chaque fois qu'on descend suivant une arête. Autrement dit, le niveau d'un nœud est toujours un de plus que le niveau de son père. Par exemple sur la figure 1, le nœud B est au niveau 2 et le nœud T est au niveau 4 ; les niveaux sont 1,2,3,4 en partant de la racine[2].

La ***hauteur*** ou ***profondeur*** d'un arbre est le nombre de niveaux de l'arbre[3]. Par exemple, l'arbre de la figure 1 est de profondeur *4*.

La ***longueur totale*** (de chemins) d'un arbre est donnée par la somme des longueurs de toutes les branches reliant les nœuds de l'arbre à la racine. Pour l'arbre de la figure 1, on trouve une longueur totale égale à 21. Si l'on distingue les nœuds internes des nœuds externes, on peut définir une ***longueur totale interne*** et une ***longueur totale externe***. Dans le même exemple, elles seraient respectivement égales à 17 et 4.

[2] Une autre convention parfois utilisée est d'atribuer le niveau *0* à la racine. Dans ce cas les niveaux de l'arbre de la figure 1 seraient 0,1,2,3.
[3] certains auteurs le définissent comme le nombre d'arêtes entre la racine et les nœuds du dernier niveau, ce qui donnerait 3 au lieu de 4 pour l'arbre de la figure 1.

Le ***degré*** d'un nœud est donné par le nombre de fils qu'il possède. Le ***degré*** d'un arbre est le degré maximum de ses nœuds. Un arbre de degré *m* est encore appelé ***arbre m-aire*** ; certains auteurs considèrent que l'adjectif ***m-aire*** doit être réservé aux arbres dont tous les nœuds non terminaux[4] sont exactement de degré *m*.

Un arbre m-aire est dit ***plein*** si seuls les nœuds du dernier niveau sont terminaux, les nœuds terminaux étant tous de degré *m*.

Le type d'arbre m-aire le plus simple est l'***arbre binaire*** pour lequel $m = 2$. Un arbre binaire est généralement considéré comme ordonné et on distinguera, pour les nœuds où ils existent tous les deux, le fils ***gauche*** et le fils ***droit***. La figure 2 montre un exemple d'arbre binaire et la figure 3 un exemple d'arbre binaire plein.

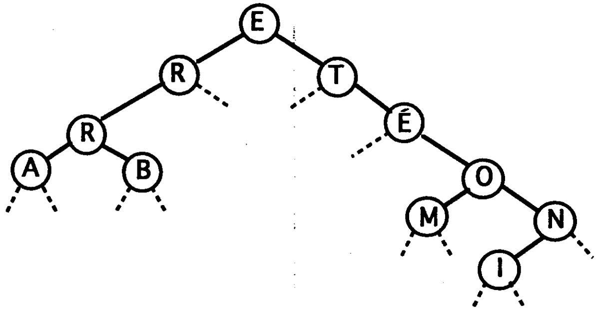

Figure 2. Un arbre binaire.

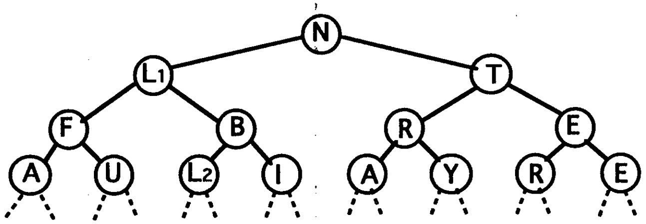

Figure 3. Un arbre binaire plein ("a full binary tree").

Les arbres présentent une relation étroite avec la récursion. La manière la plus simple de décrire un arbre est peut-être de la manière récursive suivante :

*un **arbre** est soit un nœud isolé, soit un nœud-racine relié à un ensemble d'arbres.*

et de même pour un arbre binaire :

*un **arbre binaire** est soit un nœud externe, soit un nœud interne (racine) relié à un arbre binaire gauche et à un arbre binaire droit.*

[4] Remarquons que les feuilles, quant à elles, sont toujours de degré 0.

PROPRIETES

Propriété 1. *Il existe une branche unique reliant deux nœuds quelconques d'un arbre.*

Deux nœuds quelconques ont un ***dernier ancêtre commun***, c'est-à-dire un nœud présent sur les deux branches reliant les nœuds à la racine et dont aucun fils ne partage la même particularité. Par exemple, sur l'arbre de la figure 3, L_1 est le dernier ancêtre commun à U et B. Un tel dernier ancêtre commun doit toujours exister car soit la racine joue ce rôle, soit les deux nœuds sont dans l'un des deux ***sous-arbres*** de celle-ci. Dans le dernier cas, Soit ce nœud est le dernier ancêtre recherché, soit les deux nœuds sont dans l'un des deux sous-arbres dont il constitue la racine, et ainsi de suite. Il existe donc une branche entre chaque nœud et ce dernier ancêtre commun. Enfin le recollement de ces deux branches donne le chemin unique reliant les deux nœuds choisis. ◊

Une importante conséquence de la propriété précédente est que tout nœud peut être la racine : chaque nœud d'un arbre est tel qu'il existe un et un seul chemin qui le relie à tout autre nœud du même arbre. D'un point de vue pratique, dans la définition que nous avons donnée plus haut, la racine est singularisée et l'on dit parfois de tels arbres qu'ils sont ***orientés***. Un arbre dont la racine n'est pas identifiée est une ***arborescence***.

Propriété 2. *Un arbre de N nœuds compte N-1 arêtes.*

Cette propriété se déduit directement du fait que tout nœud, sauf la racine, a un père unique et que chaque arête relie un nœud à son père. On peut aussi prouver cette propriété par récurrence à partir de la définition récursive. ◊

Les deux propriétés suivantes sont propres aux arbres **binaires.**

Propriété 3. *Pour tout arbre binaire non vide, si n_0, n_1, n_2 sont respectivements les nombres de nœuds de degré 0,1,et 2, alors* $\boxed{n_0 = n_2 + 1}$ *et ceci indépendamment de la valeur de n_1. C'est donc dire que moins de la moitié des nœuds sont de degré 2.*

Cette propriété se prouve par récurrence. Ou encore de la manière suivante : Soit N le nombre total de nœuds. $N = n_0 + n_1 + n_2$. On sait (propriété 2) que le nombre d'arêtes est $L = N - 1$. De chaque nœud part(ent) 0, 1 ou 2 arêtes suivant le degré du nœud. Donc $L = n_1 + 2n_2$. Il s'en suit que :
$N = n_0 + n_1 + n_2 = 1 + n_1 + 2n_2$. Et finalement $n_0 = n_2 + 1$. ◊

Propriété 4. *La différence entre la longueur totale externe et la longueur totale interne de tout arbre binaire dont les N nœuds internes sont tous de degré 2 est égale à 2N.*

Cette propriété se prouve aussi par récurrence, mais il est instructif d'examiner une autre démonstration. Tout arbre binaire peut être construit par le processus suivant : on commence par l'arbre binaire composé d'un nœud externe. On répète ensuite *N* fois l'action consistant à choisir un nœud externe et à le remplacer par un nouveau nœud interne avec deux nœuds-fils externes. Si le nœud externe choisi est au niveau *k*, la longueur totale interne est augmentée de *k*, mais la longueur totale externe est, elle, augmentée de *k+2* (un nœud externe du niveau *k* est supprimé, mais deux nouveaux nœuds externes sont rajoutés au niveau *k+1)*. Le processus débute sur un arbre de longueurs totales externe et interne nulles et, à chacune des *N* itérations, la longueur totale externe se voit augmentée de deux unités de plus que la longueur totale interne. ◊

Pour finir, examinons des relations entre nombre de nœuds et profondeur pour un arbre binaire.

Propriété 5. *Pour un arbre binaire de profondeur k :*

- *le nombre maximum de nœuds est $2^k - 1$.*
- *le nombre maximum de feuilles est 2^{k-1}.*
- *chacun de ces maxima est uniquement atteint pour l'arbre binaire plein.*

En se reportant à la figure 3, il est facile de montrer par récurrence que dans un arbre binaire plein le nombre de nœuds de niveau i est 2^{i-1}. Le nombre de nœuds total est donc $\sum_{i=1}^{k} 2^{i-1} = \sum_{j=0}^{k-1} 2^j = 2^k - 1$. Quant au nombre de feuilles (nœuds terminaux qui sont dans ce cas les nœuds du dernier niveau) il est égal à 2^{k-1}. Maintenant le nombre de nœuds d'un arbre binaire non plein est, de façon évidente, strictement inférieur à celui de l'arbre binaire plein. Il en est de même pour le nombre de feuilles. En effet pour passer de l'arbre binaire plein à un autre arbre binaire, on n'a besoin que de supprimer des nœuds de proche en proche (jamais en rajouter). Et à chaque étape le seul nœud qu'on peut supprimer est une feuille. Maintenant le fait de supprimer une feuille peut en créer une nouvelle, mais pas plus. Donc le nombre de feuilles ne peut que diminuer ou rester le même. Enfin la première feuille à supprimer fait obligatoirement diminuer le nombre de feuilles. ◊

Propriété 5bis. *La profondeur d'un arbre binaire possédant N nœuds est comprise entre* ***$\log_2(N+1)$*** *(cas de l'arbre binaire plein) et N (cas de l'arbre* ***linéaire*** *pour lequel chaque nœud interne a en fait un fils et un seul).*

Il est clair, d'après la propriété précédente, que pour un même nombre de nœuds la profondeur est maximale pour l'arbre linéaire (soit N) et minimale pour l'arbre plein pour lequel on a $N = 2^k-1$.

Or $N = 2^k-1 \Leftrightarrow N+1 = 2^k \Leftrightarrow k = \log_2(N+1)$ ◊

REPRESENTATION DES ARBRES BINAIRES

La représentation la plus fréquente des arbres binaires consiste en une représentation de chaque nœud par un enregistrement (structure en C) muni, en plus des informations correspondantes, de deux *liens* :

- un lien vers son ***fils gauche***
- un lien vers son ***fils droit***

Chacun de ces liens étant souvent matérialisé en fait par un *pointeur* vers le nœud fils correspondant. Lorsqu'un fils est absent, on dit souvent que le lien est ***vide***, ce qui signifie, dans le cas d'un pointeur, que celui-ci est **nul** (=**NULL** en **C**).

Remarque 1 : A un arbre binaire (dont les nœuds possèdent 0, 1 ou 2 fils) peut correspondre plusieurs représentations sous cette forme, car les fils uniques peuvent être considérés de type gauche ou droit.

Remarque 2 : Plutôt que d'utiliser des pointeurs nuls, on pointe parfois vers un type particulier de nœud externe dépourvu de descendance ; de tels nœuds sont dits "*vides*" ou "*factices*", et ne portent généralement pas de nom ni ne contiennent d'informations particulières. Dans ce cas le lien vers un tel fils n'est pas vraiment vide, mais c'est le fils lui-même qui est vide.

Remarque 3 : Cette représentation par nœud à 2 liens pour un arbre binaire de N nœuds conduit à 2N liens. Or il faut se souvenir (*Propriété 2*) que l'arbre à représenter n'a que N-1 arêtes. Ce qui veut dire que N+1 liens, soit plus de la moitié des liens, sont vides.

Pour éviter cette place perdue, deux solutions sont possibles :

- Utiliser un type de nœud différent (dépourvu de liens et éventuellement dépourvu d'information) pour les nœuds terminaux (externes). C'est plus difficile à gérer.
- Faire pointer les liens vides vers d'autre nœuds de l'arbre. Dans ce cas il sera nécessaire de marquer ces liens pour les distinguer des autres nœuds de l'arbre. On verra plus loin ce type d'arbres appelés *Arbres à brins*.

Un exemple simple d'utilisation et de construction d'arbre binaire est celui du traitement d'expressions arithmétiques. Il existe une relation fondamentale entre les expressions arithmétiques et les arbres. La figure 4 montre l'arbre syntaxique associé à une expression arithmétique écrite de façon classique (écriture ***infixée***).

On a utilisé ici des lettres de l'alphabet (une par nœud) pour identifier les opérandes et non des nombres, pour des raisons qui deviendront évidentes par la suite. ***L'arbre syntaxique*** d'une expression ***infixée*** est défini par la règle récursive simple suivante :

placer l'opérateur à la racine, puis à gauche l'arbre pour l'expression correspondant au premier opérande et à droite l'arbre pour l'expression correspondant au deuxième opérande.

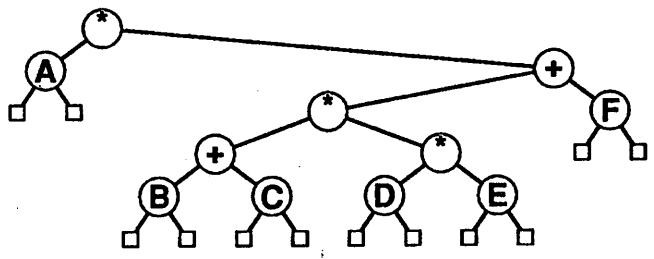

Figure 4. Arbre syntaxique de l'expression **A*(((B+C)*(D*E))+F).**

La figure 4 représente aussi l'arbre syntaxique de *A B C + D E * * F + ** qui est la même expression, mais en écriture ***postfixée***[5]. Les écritures infixée et postfixée sont deux manières de représenter des expressions mathématiques, l'écriture préfixée en est une troisième, moins courante. L'arbre syntaxique les contient toutes les trois : On retrouve chacune en parcourant l'arbre de manière convenable. (Voir plus loin : Parcours d'arbres).

Comme les opérateurs sont *dyadiques*[6] (ils acceptent deux opérandes et deux seulement), un arbre binaire est très bien adapté pour ce genre d'expression.

Le programme suivant construit l'arbre syntaxique d'une expression arithmétique à partir d'une représentation postfixée donnée en entrée. Il s'agit d'une variante légèrement modifiée d'un programme permettant l'évaluation d'expressions postfixées à l'aide d'une pile. Plutôt que des résultats intermédiaires, ce sont des pointeurs (vers les nœuds de l'arbre en construction) que l'on stocke dans la pile. On supposera donc que les procédures et fonctions **InitialiserPile, Empiler** et **Dépiler** permettent de travailler sur des pointeurs plutôt que sur des entiers. Chaque nœud porte une valeur (un caractère ici) et deux liens vers d'autres nœuds.

Construction d'un arbre syntaxique

```
struct noeud
    { char info;struct noeud *g,*d;};

struct noeud *x;
char c;
InitialiserPile();
while((c=getchar())!=EOF);)
    {
    x=(struct noeud *)malloc(sizeof(struct noeud)) ;
    if (x==NULL) {/* erreur mémoire */ exit(1);}
    x->info = c;
    x->g = NULL;   /* ou encorex->g = &z;              */
    x->d = NULL;   /*      et  x->g = &z; (voir ci-après) */
    if ( c=='+'  || c=='*' )
        { x->d = Dépiler() ; x->g = Dépiler() ; }
    Empiler(x);
    }
```

[5] cette écriture, aussi appelée *écriture polonaise inversée*, évite les parenthèses et simplifie les programmes.
[6] ou *binaires*

Commentaires sur ce programme : Tant qu'un caractère autre que le caractère <EOF> (fin de fichier) est rencontré en entrée, un nœud est créé pour l'y placer, à l'aide de la fonction d'allocation mémoire **malloc**. S'il s'agit d'un opérateur, les sous-arbres relatifs à ses opérandes sont situés aux deux premières positions dans la pile. S'il s'agit d'un opérande, ses liens sont **"nuls"**.

Au lieu de représenter un tel lien nul par un pointeur NULL, on a recours parfois à un nœud factice dont les liens pointent sur lui-même :

```
struct noeud z;
z.g = &z;
z.d = &z;
```

La figure 5 montre les étapes intermédiaires de la construction de l'arbre **syntaxique** de la figure 4 par le programme précédent. Ce programme relativement simple peut être modifié pour travailler sur des expressions plus complexes contenant des opérateurs *monadiques*[7] (à un seul opérande) comme l'exponentiation. Le mécanisme décrit est très général ; c'est exactement le même qui est **utilisé,** par exemple, pour analyser la syntaxe d'un programme en C et pour le compiler. Une fois que l'arbre syntaxique a été créé, il peut servir à de nombreux usages, comme l'évaluation de l'expression ou la création de programmes pour l'évaluer. Nous verrons plus loin comment utiliser l'arbre lui-même pour évaluer l'expression.

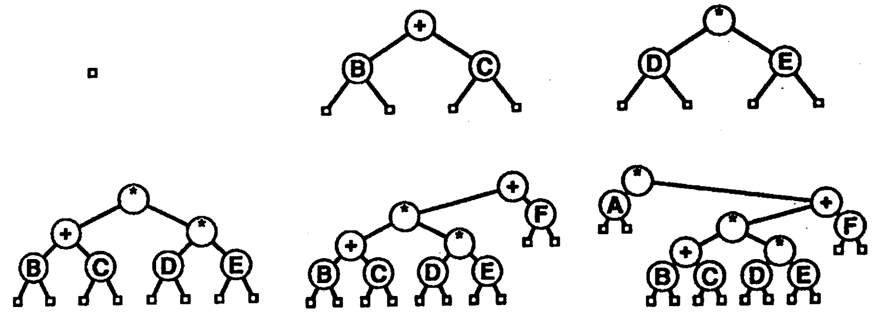

Figure 5. Construction de l'arbre syntaxique pour A B C + D E * * F + *.

Comme dans le cas de listes chaînées, il est possible de remplacer structures et pointeurs par des tableaux parallèles dans l'implantation des arbres binaires. Ceci est particulièrement recommandé si le nombre de nœuds est connu à l'avance. Et de même, le cas particulier où les nœuds doivent faire partie d'un tableau pour une autre raison impose ce modèle naturellement.

La représentation par doubles liens des arbres binaires présentée plus haut permet de *descendre* dans l'arbre, mais pas de le *remonter.* On se retrouve devant le même phénomène que l'opposition entre les listes simplement et doublement chaînées : il est

[7] on dit encore : *unaires*

possible de rajouter un lien supplémentaire à chaque nœud pour permettre une plus grande liberté de mouvement, mais au prix d'une implantation plus délicate. Diverses autres options existent parmi les structures de données plus sophistiquées pour faciliter le déplacement à l'intérieur des arbres.

REPRESENTATION D'ARBRES QUELCONQUES OU D'ARBRES GENERALISES

Représentations à nombre de liens variable ou supérieur à 2

Les arbres binaires comportent deux liens "sous" chaque nœud interne et le modèle de représentation précédent est très naturel. Mais que dire des autres arbres ou des forêts dans lesquels tout nœud peut avoir un nombre arbitraire de liens vers sa descendance ? Pour représenter un tel arbre, on pourrait utiliser des nœuds de tailles variables. Bien que ce ne soit pas impossible, c'est plutôt difficile à mettre en œuvre. Une autre manière de faire consisterait à utiliser des nœuds de taille fixe ; chacun ayant m liens, (avec m = degré de l'arbre). Cette méthode présente le défaut de gaspiller beaucoup de place. En effet :

Propriété 6. *Si un arbre m-aire possède N nœuds, alors parmi les Nm liens possibles, N(m-1) + 1 sont vides.*

Cette propriété, déjà vue pour N=2, découle trivialement de la *Propriété* 2 sur le nombre d'arêtes (c'est à dire de liens non vides). ◊

On voit donc que si dans un arbre binaire, plus de la moitié des liens sont vides, cette proportion passe à $\frac{2}{3}$ pour un arbre **ternaire**, à $\frac{3}{4}$ pour un arbre **4-aire** et finalement tend vers 1 lorsque le degré de l'arbre augmente.

Représentation par liens-père

Comme dans beaucoup d'applications il n'est pas nécessaire de descendre dans l'arbre, mais seulement de le *remonter,* on peut se contenter d'un seul lien entre tout nœud et son père. La figure 6 illustre ce modèle pour l'arbre de la figure 1 : le tableau *a* contient les informations associées à chaque enregistrement et le tableau *père* contient les liens-père. Ainsi, l'information relative au père de *a*[*i*] se trouve dans *a*[*père*[*i*]]. Par convention, on fait pointer la racine sur elle-même. Cette représentation, assez compacte, est recommandée si l'on souhaite simplement remonter l'arbre.

k	1	2	3	4	5	6	7	8	9	10	11
a[k]	A	R	B	R	E	T	E	M	O	I	N
père[k]	3	3	10	8	8	8	8	9	10	10	10

Figure 6. Représentation par liens-père d'un arbre.

Représentation par arbre binaire associé

Pour représenter un arbre quelconque dans lequel on souhaite effectuer des descentes, il est indispensable de pouvoir assumer les enfants de chaque nœud sans avoir à allouer, à l'avance, la place nécessaire à un nombre fixé de descendants. Mais c'est exactement ce pourquoi les listes chaînées ont été conçues. De manière évidente, les enfants de chaque nœud doivent être représentés comme éléments d'une liste chaînée. Chaque nœud contient alors deux liens, un pour la liste chaînée qui relie entre eux tous ses frères et lui-même, et un autre pour la liste chaînée de ses enfants. La figure 7 montre cette représentation pour l'arbre de la figure 1. Plutôt que d'utiliser un nœud factice pour terminer chaque liste, il est préférable de faire pointer le dernier nœud vers son père ; ceci donne un moyen de remonter dans l'arbre, aussi bien que le reste de la technique permet de le descendre. Il faudra alors pouvoir repérer facilement cette remontée vers le père soit en "marquant" ce dernier type d'arêtes afin de les distinguer des liens "frères", soit en marquant ou en stockant le nom du père provisoirement avant de parcourir la liste de ses enfants.

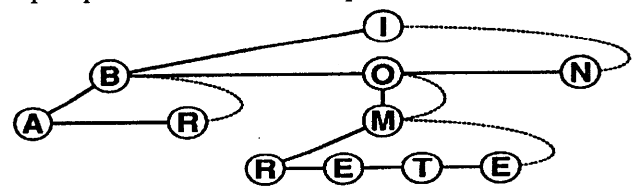

Figure 7. Représentation par fils gauche et frère droit d'un arbre.

Mais, dans cette représentation, tout nœud possède exactement deux liens (un vers son frère de droite et un autre vers son fils le plus à gauche). On peut alors se demander s'il existe une différence entre cette structure de données et un arbre binaire. La réponse est qu'il n'y en a pas, comme le montre la figure 8 qui donne la représentation binaire de l'arbre de la figure 1. Ainsi, à tout arbre peut être associé un arbre binaire obtenu en prenant comme lien gauche de tout nœud le lien vers son fils le plus à gauche dans la représentation générale, et comme lien droit le lien vers son frère de droite.

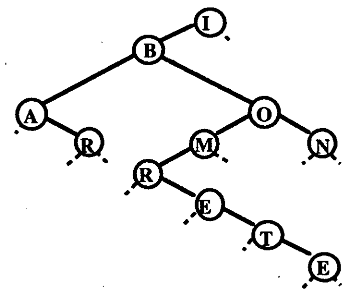

Figure 8. Représentation de l'arbre de la figure 7 par son arbre binaire associé.

En outre cette méthode permet de représenter non seulement un arbre quelconque, mais aussi une forêt. Il suffit pour cela d'imaginer un nœud supplémentaire et de le considérer comme nœud racine, et père de chacun des arbres de la forêt. Ce nœud supplementaire n'a en fait pas besoin d'être représenté dans l'arbre binaire associé, et on peut vérifier qu'un tel arbre binaire ne peut être associé qu'à une forêt (voir figure 8-bis). On parle alors parfois d'arbres généralisés.

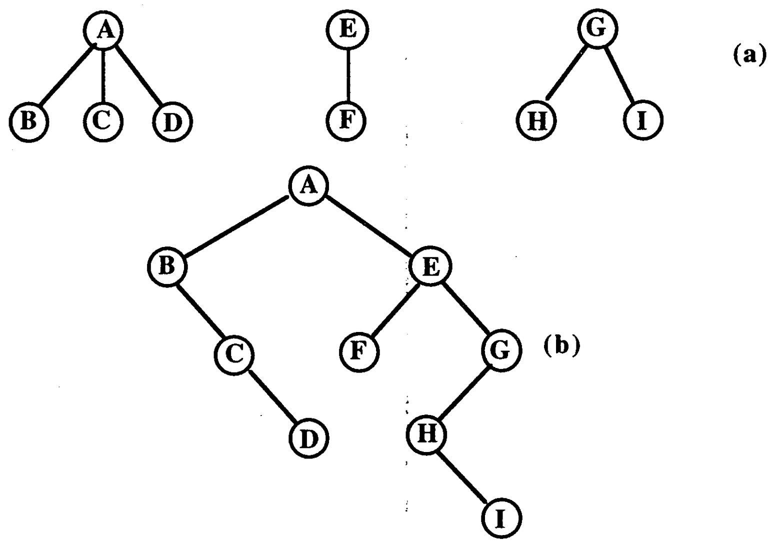

Figure 8-bis. Représentation d'une forêt (a) par son arbre binaire associé (b).

Il est ainsi possible d'utiliser n'importe quelle sorte d'arbres dans l'écriture d'un algorithme. S'il est seulement nécessaire de remonter un arbre, le lien-père permet de le faire très facilement et si on souhaite autoriser la descente, on peut toujours se ramener à un arbre binaire.

PARCOURS D'ARBRES

Une fois qu'un arbre a été construit, la première chose que l'on souhaite faire est de le ***parcourir***, c'est-à-dire visiter chacun de ses nœuds de manière systématique. Cette opération est triviale pour les listes linéaires par définition ; pour les arbres, différentes techniques se présentent, qui diffèrent principalement quant à l'***ordre*** dans lequel les nœuds sont explorés. Comme nous le verrons, chaque ordre de visite est typique de chaque application souhaitée.

Pour l'instant, intéressons-nous au parcours d'arbres binaires. Nous verrons plus loin comment adapter les méthodes décrites à n'importe quel type d'arbres, grace à la correspondance possible entre un arbre (éventuellement généralisé) et son arbre binaire associé.

Parcours préfixés d'arbres binaires

Considérons en premier l'ordre de visite ***préfixé*** qui permet d'obtenir, par exemple, l'écriture de l'expression représentée par l'arbre de la figure 4 sous forme ***préfixée.***

La méthode est définie par la loi récursive suivante :

visiter la racine, le sous-arbre gauche puis le sous-arbre droit.

On montrera ultérieurement que l'implantation récursive la plus simple de cette méthode est étroitement liée à l'implantation suivante fondée sur une pile :

Parcours préfixé : implantation itérative avec pile

```
ParcoursPréfixé(struct noeud *t)
{
 Empiler(t);
 while ( !PileVide())
    {
     t= Dépiler();Visiter (t);
     if(t->d!=z) Empiler(t->d);
     if(t->g!=z) Empiler(t->g);
    }
}
```

(La pile est supposée avoir été initialisée en dehors de la procédure.) En accord avec la règle énoncée, on "visite un sous-arbre" en visitant la racine en premier. Comme il est impossible de visiter les deux sous-arbres en même temps, on commence par placer le sous-arbre droit au sommet de la pile et l'on visite l'arbre gauche. A la fin de cette visite, le sous-arbre droit est au sommet de la pile et peut être visité.

Les figures 9 à 12 suivantes représentent toutes l'arbre de la figure 2. A vous de parcourir cet arbre dans l'ordre indiqué.

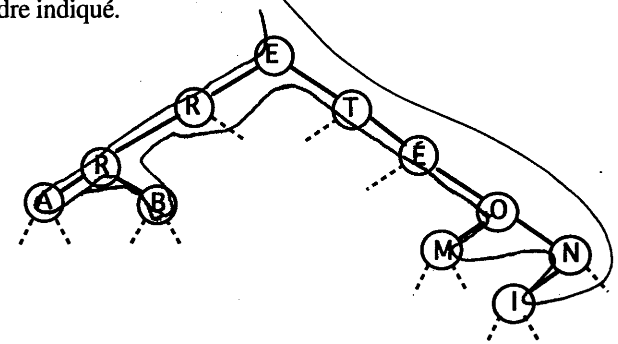

Figure 9. Arbre de la figure 2 : → Parcours PREFIXE.

Dans le cas de l'arbre syntaxique de la figure 4, on obtient l'ordre suivant, qui n'est autre que l'écriture préfixée de l'expression :

* A + * + B C * D E F

que l'on peut sans doute lire plus aisément en lui rajoutant des parenthèses :

* A(+(*(+ B C)(* D E))F)

Pour se prouver que le programme exécute effectivement un parcours préfixé des nœuds de l'arbre, on peut utiliser un raisonnement par récurrence avec comme hypothèse de récurrence que les sous-arbres sont parcourus dans l'ordre préfixé *et* que le contenu de la pile avant et après le parcours d'un sous-arbre est le même.

Parcours infixés d'arbres binaires

La deuxième technique est le parcours ***infixé*** qui peut être utilisé, par exemple, pour écrire, à partir d'arbres syntaxiques, des expressions arithmétiques sous forme ***infixée*** (moyennant un petit travail supplémentaire pour insérer des parenthèses aux bons endroits). Le parcours infixé de l'arbre de la figure 4 donnera ainsi l'expression telle qu'elle est écrite sur la légende de cette figure. De la même façon que précédemment, le parcours ***infixé*** est caractérisé par une loi récursive simple :

visiter le sous-arbre gauche, la racine puis le sous-arbre droit.

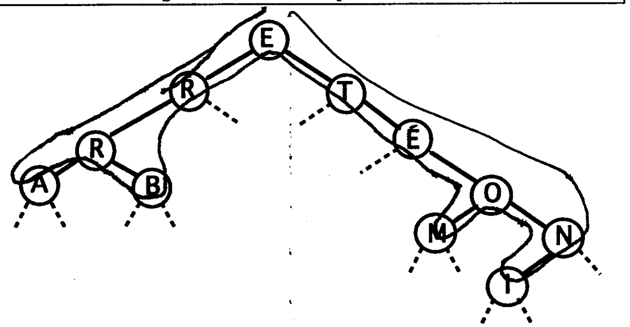

Figure 10. Arbre de la figure 2 : → Parcours INFIXE.

On parle parfois aussi de parcours *symétrique* pour des raisons évidentes. Ce type de parcours est probablement le plus répandu. L'implantation d'un parcours infixé à partir d'une pile est pratiquement identique à la précédente. La figure 10 doit conduire à la succession A R B R E T É M O I N des nœuds dans le parcours infixé de l'arbre de la figure 2.

Parcours postfixés d'arbres binaires

Le troisième type de parcours récursif, le parcours ***postfixé***, est bien évidemment défini par la loi récursive :

visiter le sous-arbre gauche, le sous-arbre droit puis la racine.

Le parcours postfixé de l'arbre de la figure 2 donne (figure 11) la succession A B R R M I N O É T E des nœuds. Le parcours postfixé de l'arbre de la figure 4 donne quant à lui l'expression A B C + D E * * F + *, comme on pouvait s'y attendre. L'implantation d'une représentation à base de pile d'un parcours postfixé est plus complexe que les deux précédentes car on doit s'arranger pour empiler la racine et le sous-arbre droit pendant que le sous-arbre gauche est exploré *et* pour empiler la racine pendant que le sous-arbre droit est exploré. Les détails de cette implantation sont laissés en exercice au lecteur.

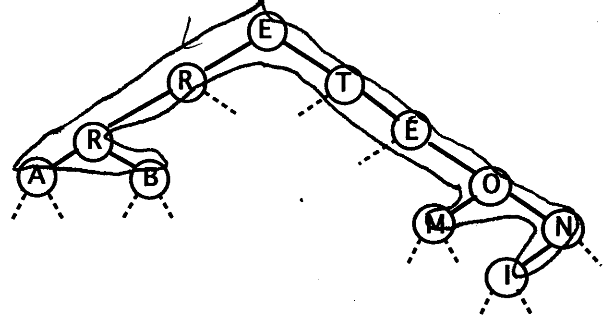

Figure 11. Arbre de la figure 2 : → Parcours POSTFIXE.

Parcours par niveaux d'arbres binaires

La dernière stratégie de parcours n'est pas du tout récursive : on visite simplement les nœuds dans l'ordre où ils apparaissent dans la représentation graphique, en allant du haut vers le bas et de gauche à droite. Cette technique porte le nom de ***parcours par niveaux*** car tous les nœuds d'un même niveau sont visités d'affilée, dans l'ordre gauche-droite. La figure 12 illustre l'ordre de visite par niveau des nœuds de l'arbre de la figure 2.

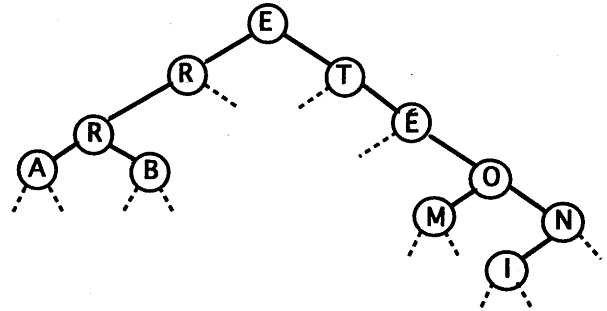

Figure 12. Arbre de la figure 2 : → Parcours PAR NIVEAUX.

D 'une manière remarquable, le parcours par niveau peut être effectué en utilisant une file à la place d'une pile (et de commencer par le fils gauche) dans le programme de parcours préfixé :

Parcours par niveau

```
ParcoursNiveau(struct noeud *t)
{
 Enfiler(t) ;
 while(!FileVide())
    {
     t=Défiler();
     Visiter(t);
     if (t->g != z) Enfiler (t->g);
     if (t->d != z> Enfiler (t->d);
    }
}
```

Si, d'un côté, les deux programmes sont très semblables (la seule différence étant que l'un utilise une structure LIFO[8] et l'autre une structure FIFO[9]), de l'autre côté, ils traitent les arbres de manières fondamentalement différentes. Ces programmes sont intéressants car ils matérialisent la différence essentielle entre les structures de pile et de file.

Le parcours par niveaux permet d'autre part la numérotation des nœuds par niveaux, qui nous sera utile plus loin pour définir les arbres binaires complets (figures 15 et 16).

Parcours d'arbres quelconques ou généralisés

Les parcours préfixé, postfixé et par niveau sont aussi bien définis pour les arbres quelconques, voire généralisés. La loi de parcours préfixé et de parcours postfixé deviennent respectivement :

visiter la racine puis chacun des sous-arbres
visiter chacun des sous-arbres puis la racine

Il est difficile de donner un sens à un ordre infixé dans le cas d'arbres autres que binaires. On pourrait peut-être dire qu'on visite la racine juste après avoir visité le premier sous-arbre. Mais cela nécessiterait que l'arbre soit ordonné, et que l'on privilégie ce premier sous-arbre.

La loi de parcours par niveau est identique à celle des arbres binaires.

Terminons par deux propositions permettant de relier le parcours d'un arbre quelconque à celui de son arbre binaire associé, tel que celui-ci a été défini plus haut :

Proposition 1. *Parcourir un arbre quelconque (ou généralisé) en ordre* ***préfixé*** *revient à parcourir l'arbre binaire associé dans l'ordre* ***préfixé****.*

Proposition 2. *Parcourir un arbre quelconque (ou généralisé) en ordre* ***postfixé*** *revient à parcourir l'arbre binaire associé dans l'ordre* ***infixé****.*

La démonstration de ces deux propositions découle directement de la comparaison des définitions récursives des ordres de parcours et du fait que les sous-arbres d'un nœud quelconque sont représentés par le sous-arbre gauche dans l'arbre binaire associé, tandis que les autres sous-arbres de son père (ses sous-arbres frères) sont représentés par le sous-arbre droit dans le même arbre binaire associé. ◊

On obtient alors directement, à partir des programmes précédents pour les arbres binaires, des implantations à base de piles et de files pour les parcours des arbres généralisés.

Par contre il n'y a aucune correspondance entre les parcours par niveaux, ni avec le parcours postfixé de l'arbre binaire associé.

ARBRE BINAIRES À BRINS (Threaded binary trees)

Nous avons dit (*Remarque 3*, *Propriété 6*) que plus de la moitié des liens d'un arbre binaire sont vides. Une idée intelligente pour utiliser ces liens a été proposée par A.J.Perlis et C.Thornton. Elle consiste à remplacer ces liens par des pointeurs (appelés ***brins*** ou ***fils***[10]) vers d'autres nœuds de l'arbre. Si le pointeur de **droite** [resp[t] de **gauche**] du nœud π est normalement **nul**, nous le remplacerons par un pointeur vers le nœud qui serait visité **après** [resp[t] **avant**] le nœud π dans l'ordre **infixé**.

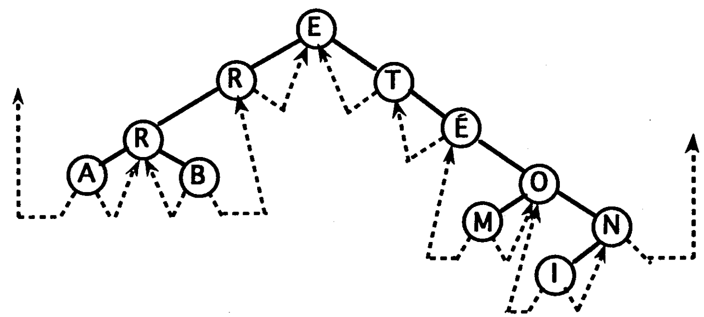

Figure 13. Un exemple d'arbre à brins.

De cette manière le parcours infixé de l'arbre se trouve très simplifié[11] : On part du nœud le plus à gauche puis on suit les liens de droite en pensant à redescendre à gauche si on trouve de vrais liens fils-gauche (nœud O sur l'exemple). Il n'y a donc plus besoin de pile. Nous devons cependant être capable de distinguer les **brins** ainsi créés des pointeurs normaux. Il suffit pour cela d'ajouter deux champs (membres) booléens à la structure représentant un nœud. Ils occuperont peu de place (un octet chacun, voire un bit chacun).

donnée	pointeur gauche	pointeur droit	bring	brind
			vrai ou faux	vrai ou faux

[8] Last In First Out = Dernier Entré, Premier Sorti = pile
[9] First In First Out = Premier Entré, Premier Sorti = file

[10] ***thread*** en anglais. Le mot ***brin*** est préférable pour éviter la confusion avec les ***fils*** (***enfants***) de chaque nœud.
[11] il se trouve que le parcours préfixé est également plus simple.

les fils ont pour n° 2n & 2n+1

On peut remarquer aussi sur l'exemple de la figure 13 que deux brins restent libres : le brin gauche partant de A et le brin droit partant de N. On peut alors rajouter un nœud supplémentaire, dit ***nœud d'entête*** ("head node") qui aura pour fils la racine de l'arbre et lui-même et vers lequel pointerons ces deux brins. (voir figure 14 ci-dessous)

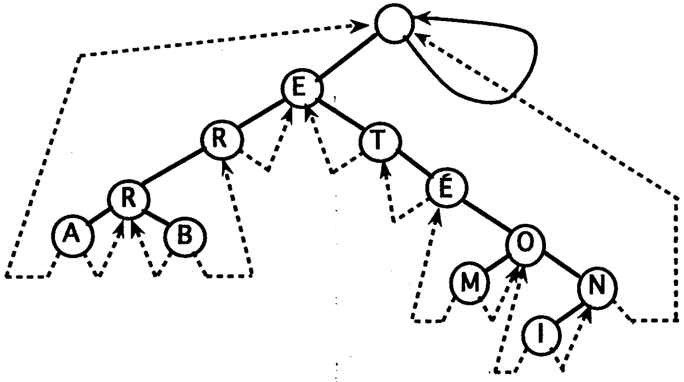

Figure 14. Arbre à brins avec nœud d'entête.

ARBRES BINAIRES COMPLETS et TAS (ou arbres maximiers)

Arbres binaires complets

Considérons un arbre binaire plein dont les nœuds sont numérotés par niveaux (figure 15) et supposons qu'on décide de ne conserver que les N premiers nœuds de cet arbre (figure 16). Un tel arbre vérifie alors les propriétés suivantes :

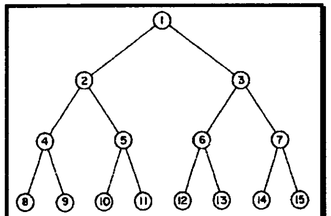

Figure 15. Arbre binaire plein numéroté par niveaux.

- $\forall i = 2$ à N, le père du nœud i est le nœud $\left\lfloor \frac{i}{2} \right\rfloor$.
- $\forall i = 1$ à $\left\lfloor \frac{N}{2} \right\rfloor$, le fils gauche du nœud i est le nœud $2i$.
- $\forall i = 1$ à $\left\lfloor \frac{N}{2} \right\rfloor$, le fils droit du nœud i est le nœud $2i+1$

  (en remarquant que pour $i=\left\lfloor \frac{N}{2} \right\rfloor$, ce fils droit n'existe que si N est impair).

- Le nœud i est interne $\Leftrightarrow i \leq \left\lfloor \frac{N}{2} \right\rfloor$.

Un tel arbre est appelé ***arbre binaire complet***.

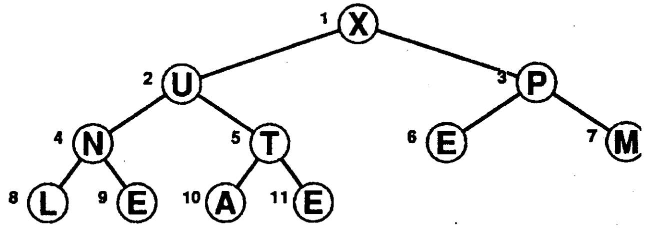

Figure 16. Un arbre binaire complet.

On peut alors constater que les relations indicielles précédentes entre pères et fils jouent le rôle de liens implicites. il n'est donc plus nécessaire d'expliciter ces liens sous la forme de pointeurs. En fait un arbre binaire complet peut être représenté par un simple tableau contenant les informations (valeurs, clés ou autres) que l'on veut attacher aux différents nœuds.

k	1	2	3	4	5	6	7	8	9	10	11
t[k]	X	U	P	N	T	E	M	L	E	A	E

Figure 17. Représentation d'un arbre binaire complet par un simple tableau.

Tas

Si en outre les données de l'arbre sont telles que la clé de chaque nœud est **supérieure** [resp[t] **inférieure**] aux clés de ses enfants, l'arbre est alors appelé arbre ***maximier*** [resp[t] ***minimier***] ou encore ***tas*** ("***heap***" en anglais). L'intérêt de cette structure est, outre l'économie de place due à la structure d'arbre binaire complet, de permettre une implantation aisée de ***files de priorité*** qui sont une généralisation des structures de **files** ou **piles** dans laquelle le premier élément à utiliser (à sortir) est celui qui a la plus grande priorité. En effet on peut montrer qu'un **tas** permet d'implanter de façon performante (en ordre de **log(N)** les 5 fonctions suivantes de gestion des files de priorité :

- *Insertion* d'un nouvel élément.
- *SupprimMax* Suppression de l'élément maximal.
- *Substitution* de l'élément maximal par un nouvel élément.
- *Modification* d'un élément et donc éventuellement de sa priorité.
- *Suppression* d'un élément quelconque désigné.

Seules les deux autres fonctions de *Construction* (en **nLog(N)**) d'une file de priorité et de *Fusion* de deux files de priorité en une seule sont moins performantes.