

# ARBRES ET FORETS

## 1. DEFINITIONS ET PROPRIETES

### 1.1. Forêts, arbres généraux et binaires

Définitions. - Une **forêt** sur un ensemble  $E$  est un graphe  $(E, \Gamma)$  sans circuit tel que tout sommet a au plus un prédécesseur : pour tout  $x \in E$ ,  $\Gamma^{-1}(x)$  a au plus un élément.

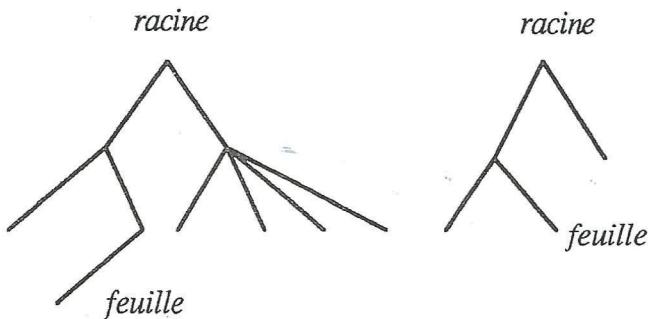


Figure 1. Un exemple de forêt

Dans la suite, à la place de **sommet**, on dira aussi **nœud** ou **point**.

On appelle **racine** un sommet  $x$  sans prédécesseur :  $\Gamma^{-1}(x) = \emptyset$ . On appelle **feuille** un sommet  $x$  sans successeur :  $\Gamma(x) = \emptyset$ .

On appelle **arborescence**, ou **arbre**, une forêt à une seule racine. On peut toujours ramener une forêt à un arbre en "enracinant" ses arborescences. On assimile souvent un nœud à l'arbre dont il est racine.

On appelle **fils de  $x$** , un élément de  $\Gamma(x)$ . On appelle **père de  $x$**  l'élément  $\Gamma^{-1}(x)$  s'il existe. Deux éléments de même père sont dits **frères**. Il est courant d'ordonner les frères, ce que l'on sous-entend généralement en les dessinant de gauche à droite dans un plan.

Le noeud  $x$  est un **ascendant** (resp. un **descendant**) du noeud  $y$  si  $x$  est père (resp. fils) de  $y$  ou ascendant (resp. descendant) du père (resp. fils) de  $y$ .

On appelle **branche** d'un arbre tout chemin de la racine à une feuille de l'arbre.

Une forêt est **étiquetée** par un ensemble  $T$  d'étiquettes quand on a une fonction de valuation  $\text{val} : E \rightarrow T$ .

On peut aussi étiqueter (valuer comme dans les graphes) les arcs, mais on ne le fera pas ici.

Définitions. - Un **arbre binaire** est un arbre tel que chaque sommet a au plus 2 fils : **droit et gauche**.

Dans une représentation planaire, on distingue soigneusement le fils gauche du fils droit : l'inclinaison est importante (fig. 2). On distingue alors aussi **sous-arbres** droit et gauche d'un sommet.

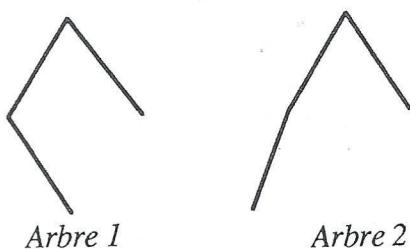


Figure 2. Deux arbres binaires distincts

On appelle **bord gauche** (resp. droit) d'un arbre binaire le chemin obtenu en parcourant exclusivement les fils gauches (resp. droits) à partir de la racine.

On peut toujours transformer un arbre quelconque en arbre binaire. Il suffit pour cela de transformer les liaisons *premier fils* (pf) et *frère* (fr) de l'arbre initial respectivement en *fils gauche* (fg) et *fils droit* (fd) de l'arbre binaire (fig. 3). Ceci peut être généralisé aux forêts si l'on considère que les racines sont frères.

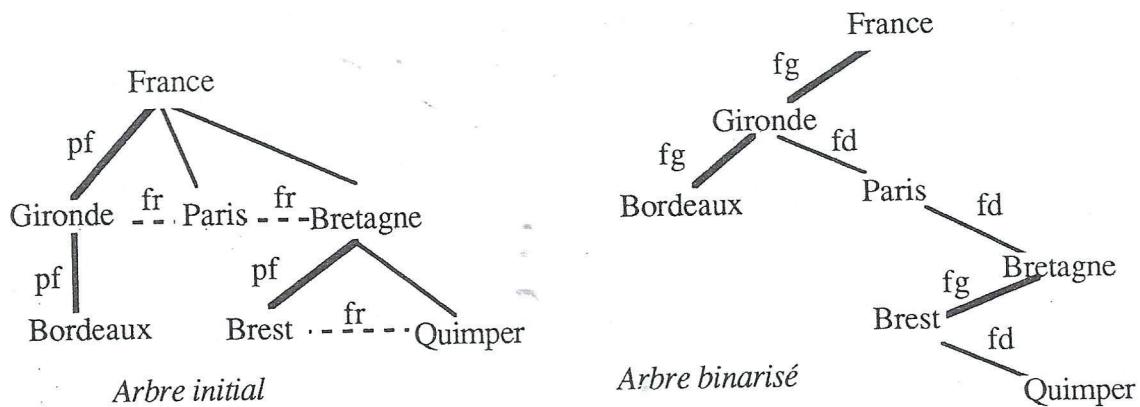


Figure 3. Binarisation d'un arbre

On peut désigner de manière unique un noeud d'un arbre en le codant par un mot sur l'ensemble des entiers naturels  $\mathbb{N}$ , appelé **occurrence** du noeud dans l'arbre, et défini par : l'occurrence de la racine est le mot vide et, si un noeud a comme occurrence  $m$ , alors son premier fils a comme occurrence  $m \cdot 0$ , le deuxième  $m \cdot 1$ , etc.

## 1.2. Hauteurs, arbres complets, arbres parfaits

**Définitions.** La hauteur (ou niveau, profondeur)  $h(x)$  d'un noeud  $x$  d'un arbre est définie par :

$$h(x) = 1 \text{ si } x \text{ est racine,}$$
$$\text{et } h(x) = 1 + h(y) \text{ si } y \text{ est le père de } x.$$

La hauteur ou profondeur d'un arbre a non vide est :

$$h(a) = \max \{h(x); x \text{ est un noeud de } a\}.$$

On pourra par la suite considérer que celle d'un arbre vide est 0.

La longueur de cheminement d'un arbre a est :

$$lc(a) = \sum h(x), x \text{ noeud de } a.$$

La longueur de cheminement externe est :

$$lce(a) = \sum h(x), x \text{ feuille de } a.$$

La longueur de cheminement interne est :

$$lci(a) = \sum h(x), x \text{ noeud interne (non feuille) de } a.$$

On a bien sûr :  $lc(a) = lce(a) + lci(a)$ .

En divisant chacune de ces quantités par le nombre de noeuds concernés, on obtient des profondeurs moyennes.

**Définitions.-** Un arbre binaire complet est un arbre binaire dont tous les noeuds qui ne sont pas des feuilles ont exactement deux fils (fig. 4).

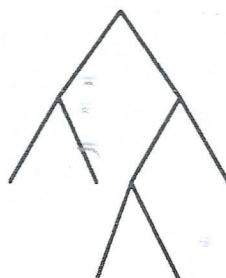


Figure 4. Arbre binaire complet

Un arbre binaire parfait est un arbre binaire dont toutes les feuilles sont situées sur au plus deux niveaux consécutifs : l'avant-dernier niveau est complet et les feuilles du dernier niveau sont groupées le plus à gauche possible (fig. 5).

Un arbre binaire parfait n'est pas nécessairement complet.

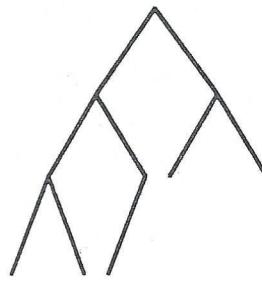


Figure 5. Arbre binaire parfait

**Lemme 1.-** Pour un arbre binaire de hauteur  $p$  ayant  $n \geq 1$  noeuds au total, on a :

$$\lfloor \log_2(n + 1) \rfloor \leq p \leq n.$$

**Preuve.** Pour  $n$  fixé, la hauteur est maximum pour des arbres dégénérés, ou *filiformes*, donc  $p \leq n$ . Pour  $n$  fixé, elle est minimale pour un arbre parfait dans ce cas on a  $n = 1 + 2 + 2^2 \dots + 2^{p-2} + f$ , où  $f$  est le nombre des feuilles tel que  $1 \leq f \leq 2^{p-1}$ . D'où  $n \leq 2^p - 1$  pour tout arbre binaire et le résultat.

**Corollaire.-** Tout arbre binaire ayant  $f$  feuilles a une hauteur  $p$  telle que  $p \geq \lceil \log_2(f) + 1 \rceil$ .

## 2. UTILISATION DES ARBRES

Nous donnons ici quelques exemples d'utilisation des arbres :

- **Questionnaire dichotomique** : faune, flore... (fig.6). Les  $q_i$  sont des questions posées par la machine, avec des réponses oui/non de l'utilisateur, et les  $r_i$  des réponses de la machine.

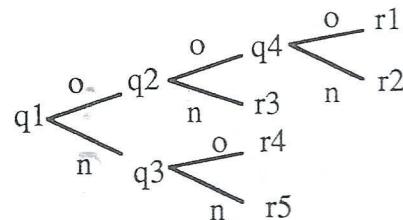


Figure 6. Un exemple de questionnaire dichotomique

- **Analyse d'une phrase** : "le concierge monte le courrier" (fig. 7).

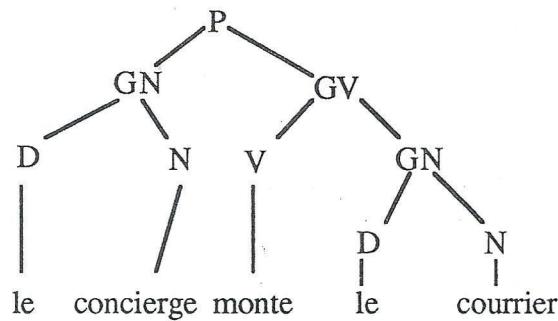


Figure 7. Arbre syntaxique d'une phrase  
 (Lexique : P : phrase ; GN : groupe nominal ; GV : groupe verbal ; D : déterminant ; N : nom ; V : verbe)

- Expression arithmétique :  $(a + b/c) * (d - 3 * e)$  (fig. 8).

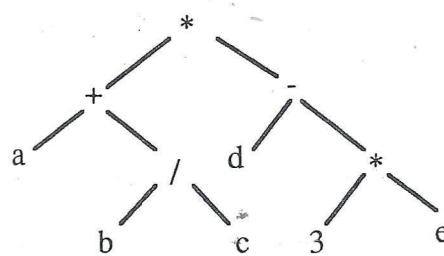


Figure 8. Arbre syntaxique abstrait d'une expression arithmétique

- Système de fichiers arborescent : exemple du système UNIX (fig. 9)

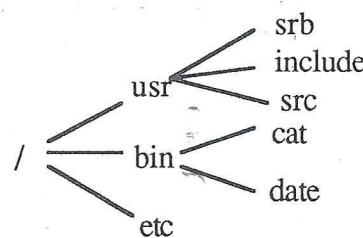


Figure 9. Une arborescence d'UNIX

- Analyse d'un programme en langage évolué :

Le programme PASCAL suivant lit 3 nombres et est censé écrire leur maximum. Il est correct syntaxiquement, mais pas sémantiquement. De plus, il n'est pas très clair car max change de signification dans le corps du programme.

```

program rechmax ;
var
  a, b, c, max : integer ;
begin
  readln (a, b, c) ;
  if a < b then max := b else max := a ;
  if c < max then max := c;
  writeln (max)
end.

```

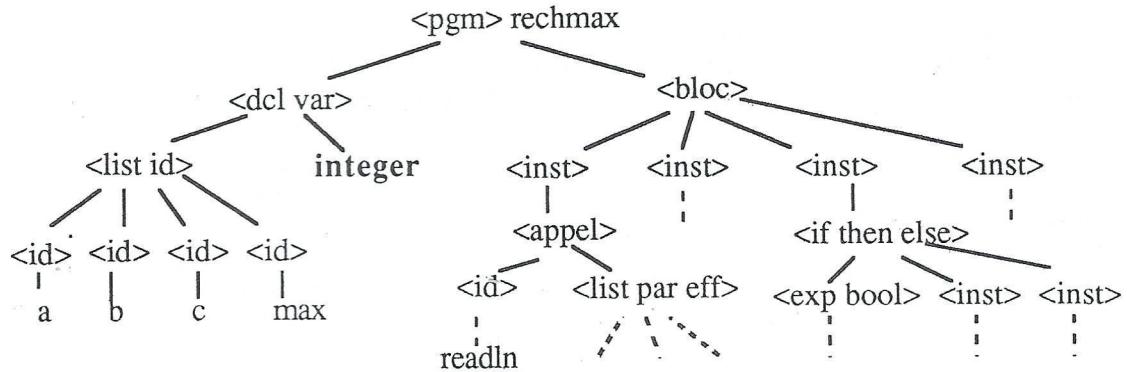


Figure 10. Un arbre syntaxique abstrait du programme ci-dessus

Des arbres syntaxiques de formes diverses, tel celui de la figure 10, sont utilisés dans les éditeurs de programmes ou les traducteurs de langages.

### 3. SPECIFICATION ALGEBRIQUE DES ARBRES BINAIRES ETIQUETES

Une spécification d'un constructeur de types abstraits d'arbres binaires (sorte arbin), où la sorte paramètre s symbolise un type d'étiquettes, peut être écrite comme suit, où  $\omega$  est une constante de T représentant une étiquette fictive :

sorte arbin (sorte s tq $\omega : \rightarrow s$ )	/* ou arbin de s */
utilise booléen	
opérations	
arbnou : $\rightarrow$ arbin	/* arbre nouveau, ou vide, noté aussi $\Lambda$ */
enrac : $s \times \text{arbin} \times \text{arbin} \rightarrow$ arbin	/* enracinement, noté aussi e */
ag, ad : arbin $\rightarrow$ arbin	/* sous-arbres gauche et droit */
rac : arbin $\rightarrow s$	/* étiquette à la racine, notée aussi r */
vide : arbin $\rightarrow$ booléen	/* test de vacuité, noté aussi v */

#### axiomes

$$\begin{aligned}
\text{rac } (\wedge) &= \omega \\
\text{vide } (\wedge) &= \text{vrai} \\
\text{rac } (\text{enrac } (x, a_1, a_2)) &= x \\
\text{vide } (\text{enrac } (x, a_1, a_2)) &= \text{faux}
\end{aligned}$$

$\text{ag}(\wedge) = \wedge$   
 $\text{ad}(\wedge) = \wedge$   
 $\text{ag}(\text{enrac}(x, a1, a2)) = a1$   
 $\text{ad}(\text{enrac}(x, a1, a2)) = a2$

### pré-conditions

$\text{pré}(\text{enrac}(x, a1, a2)) \equiv x \neq \omega$

On peut avoir bien d'autres opérations, comme la comparaison de 2 arbres, la feuille la plus à gauche, à droite, etc. Certaines seront étudiées par la suite, mais on peut déjà donner le parcours d'un arbre binaire, cas particulier du parcours de forêts.

## 4. PARCOURS DE FORETS ET D'ARBRES

### 4.1. Parcours généraux

Puisque les forêts et les arbres sont des graphes particuliers, ils sont susceptibles de subir au moins les parcours généraux qui ont été décrits dans le chapitre sur les graphes : en profondeur d'abord et en largeur d'abord.

Tout dépend en fait des liaisons qui sont effectivement représentées : père-fils, frère droit, fils-père, cousin droit du même degré le plus proche, etc.

Il faut donc bien savoir de quelle relation on parle pour choisir le type de parcours. On peut ainsi réaliser des parcours en profondeur d'abord, en largeur d'abord, par niveaux, en remontant, etc., et de plusieurs types dans chacune de ces catégories.

### 4.2. Parcours d'arbres binaires

Si l'on s'en tient à la spécification donnée précédemment, il est possible de parcourir les nœuds d'un arbre binaire principalement dans trois ordres différents, pour leur appliquer un traitement matérialisé par une procédure  $\text{pr}(x)$ , où  $x$  est l'étiquette du noeud rencontré (fig. 11) :

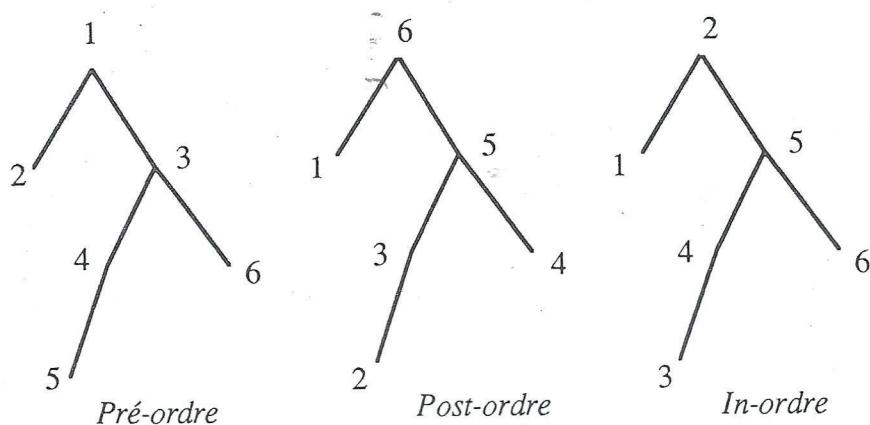


Figure 11. Parcours d'arbre binaire  
(Les numéros des noeuds donnent l'ordre du traitement)

- *pré-ordre* (ou ordre *préfixe*) : on traite récursivement d'abord la racine, puis le sous-arbre gauche, et enfin le sous-arbre droit ;
- *in-ordre* (ou ordre *infixe*) : on traite récursivement d'abord le sous-arbre gauche, puis la racine, et enfin le sous-arbre droit ;
- *post-ordre* (ou ordre *postfixe*) : on traite récursivement d'abord le sous-arbre gauche, puis le sous-arbre droit, et enfin la racine ;

De ces trois ordres résultent trois procédures de parcours d'un arbre a pour appliquer en chaque noeud une procédure pr : préfix, postfix et infix (a, pr).

Par exemple, une définition récursive de préfix est :

$\text{prefix } (a, \text{pr}) = \begin{cases} \text{résultat} = \text{si } \neg \text{vide}(a) \\ \quad \text{alors } \{\text{pr}(\text{rac}(a)), \text{prefix}(\text{ag}(a), \text{pr}), \text{prefix}(\text{ad}(a), \text{pr})\} \end{cases}$

Les autres sont similaires. Ces procédures interviennent de manière courante en informatique.

**Exemple.** - Mise d'une expression arithmétique arborescente sous forme préfixée (ce qui relève de l'**analyse syntaxique** : voir compilation), puis évaluation de cette expression par un parcours en ordre post-fixe.

Au lieu d'une procédure x-fix, on peut définir une fonction x-fix : arbin de s → liste de v, où pr est aussi une fonction pr : s → v , par exemple :

$\text{prefix } (a, \text{pr}) = \begin{cases} \text{si } \text{vide}(a) \text{ alors } \Lambda \\ \quad \text{sinon } \text{adjt}(\text{pr}(\text{rac}(a)), \text{conc}(\text{prefix}(\text{ag}(a), \text{pr}), \text{prefix}(\text{ad}(a), \text{pr}))) \end{cases}$

## 5. REPRESENTATION GENERALE DES FORETS ET ARBRES ETIQUETES

Puisque les arbres sont des graphes, on peut utiliser les représentations déjà étudiées pour ceux-ci. Cependant, il est souvent plus commode et plus efficace d'adopter des représentations particulières.

### 5.1. Représentation par chaînage pour les arbres binaires

Un arbre binaire est ici un pointeur sur un nœud. En C on peut écrire :

```
typedef struct snode { T r ; struct snode *g, *d ; } node, *arbin ;
```

où r est l'étiquette à la racine, g et d des pointeurs, respectivement sur le fils gauche et le fils droit du nœud.

**Exemple.** La représentation de l'expression arithmétique  $(1 + a / (- b)) * (c - 3) * d$  est donnée à la figure 12.

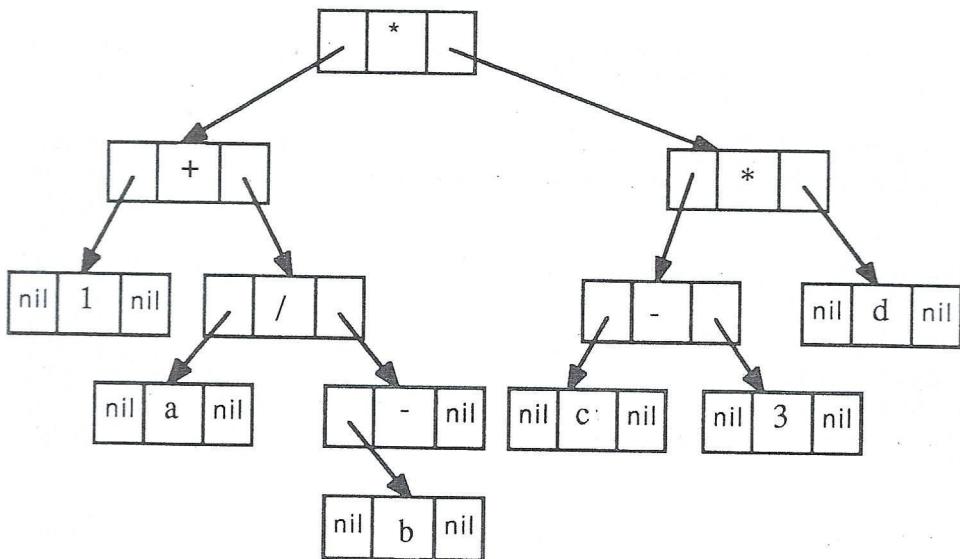


Figure 12. Une représentation chaînée d'expression

La programmation des différentes opérations est comme d'habitude réalisée de manière statique ou dynamique. Par exemple :

- l'*enracinement* enrac dynamique consiste simplement à créer un noeud, à remplir ses trois champs et à renvoyer le pointeur sur le nouveau noeud ;
- l'*arbre gauche* ag (ou l'*arbre droit* ad) peut être obtenu simplement par une fonction renvoyant un pointeur ; mais cette opération peut aussi avoir pour effet de détruire le noeud au sommet ou l'*arbre droit*, c.à d. de récupérer la place qu'ils occupent.

## 5.2. Représentation des forêts par pointeurs sur les fils

C'est une représentation tout à fait analogue à celle ci-dessus : à chaque nœud, on associe un tableau f de p pointeurs sur ses fils. Il est clair que cette représentation ne convient que si p est borné. On a alors :

```
typedef struct snode {T r; struct snode *f[p];} noeud, *forêt;
```

Ce type d'organisation est utilisé par exemple dans les systèmes de gestion de bases de données relationnelles, pour le traitement hiérarchisé d'index sous forme d'arbres de Bayer, ou *B-arbres*.

## 5.3. Représentation par chaînage pour les forêts générales binarisées

Si l'on effectue la transformation des forêts générales en arbres binaires vue précédemment, on peut utiliser directement la représentation de la section 4.1.

*Exemple.* Classification des régions et villes françaises vue précédemment.

On représente alors en C le type forêt de T, des forêts générales étiquetées par T, par :

```
typedef struct snode {T r; struct snode *fg, *fr;} noeud, *forêt;
```

où r est l'étiquette à la racine, fg et fr des pointeurs, respectivement sur le fils gauche et le frère droit du nœud.

Il est possible de représenter d'autres liens selon les parcours que l'on désire : père, frère gauche, cousin de même niveau, etc. Dans chaque cas, on prend un pointeur supplémentaire.

#### 5.4. Représentation par un tableau

On peut représenter un arbre binaire ou une forêt générale "binarisée" dans un espace de travail représenté par un tableau : noeud v[n]. Les liens peuvent être représentés par des pointeurs, en fait des indices de tableaux ou une valeur particulière pour **nil**, exactement comme ci-dessus. Il peut coexister plusieurs arbres dans cet espace, et l'espace libre peut être géré sous la forme d'une liste libre.

On peut aussi dans certains cas représenter un des liens par contiguïté, à condition de marquer son absence de manière particulière.

**Exemple.** Représentation de l'expression arithmétique  $(a + b / c) * ((-3) - e)$ , en représentant le lien gauche par contiguïté, et le lien droit par chaînage, où **nil** est symbolisé par -1, et "être une feuille" par -2 (fig. 13).

	r	d
0	*	6
1	+	3
2	a	-2
3	/	5
4	b	-2
5	c	-2
6	-	9
7	-	-1
8	3	-2
9	e	-2

Figure 13. Représentation d'une expression arithmétique par un tableau

```
typedef struct {T r ; int d ;} noeud ;
typedef int arbin ;
```

## 5.5. Représentation contiguë des arbres binaires parfaits

Pour représenter un arbre binaire parfait, on peut utiliser un entier et un tableau :

`int n : nombre de noeuds`

`T v [n] : tableau des étiquettes des noeuds.`

Pour un arbre a, on range les étiquettes dans `v` niveau par niveau :

`rac (a) dans v [0]`

`rac (ag(a)) dans v [1]`

`rac (ad(a)) dans v [2]`

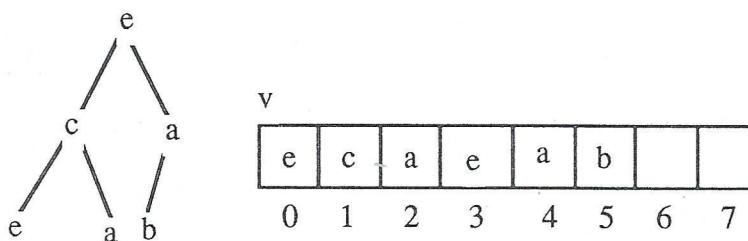


Figure 14. Arbre binaire parfait et sa représentation contiguë

D'une manière générale, pour un sous-arbre b (fig. 14) :

`rac (b) dans v [i]`

`rac (ag(b)) dans v [2i + 1]`

`rac (ad(b)) dans v [2i + 2]`

Si un sous-arbre n'existe pas, on met **nil** à la place correspondante.

On a ainsi un procédé très simple de **descente** et de **remontée** dans un tel arbre, respectivement en multipliant et en divisant par 2. Cette représentation est utilisée dans certains tris.

## 6. SUPPRESSION DE LA RECURSIVITÉ DANS LES PARCOURS D'ARBRES BINAIRES

On prend l'exemple d'une procédure de parcours en **ordre infixé**. Pour alléger, on supprime le paramètre `pr` :

`infix (a) = {résultat = si a ≠ & alors {infix (ag (a)), pr (rac (a)), infix (ad (a))}}`

La figure 11 donne l'ordre dans lequel les noeuds sont visités. Ce problème a déjà été étudié dans le cas général des graphes. Mais on peut raisonner plus directement ici.

Il faut en fait construire la **liste des noeuds** en ordre infixé et pour cela pouvoir **remonter** dans l'arbre, donc garder les références aux sous-arbres dans lesquels on est descendu : on peut utiliser une **pile p** dans le cas général. Il faut aussi pouvoir trouver la tête `t` et éventuellement la queue `q` de la liste des noeuds.

## 7. SUPPRESSION DE LA RECURSIVITE DOUBLE

On se ramène à un parcours d'arbre.

*Exemple.* Tours de Hanoï. Une procédure H résolvant ce problème pour n disques, une tour de départ a, intermédiaire b et d'arrivée c, est :

$$H(n, a, b, c) = \{ \text{résultat} = \text{si } n \neq 0 \text{ alors } \{ H(n-1, a, c, b), D(a, c), H(n-1, b, a, c) \} \};$$

où D désigne une procédure de déplacement du disque au sommet d'une tour a sur le sommet d'une tour c.

Considérons l'analyse des appels de la figure 15. On a un parcours d'arbre en ordre infixe. On peut donc supprimer la récursivité comme indiqué précédemment.

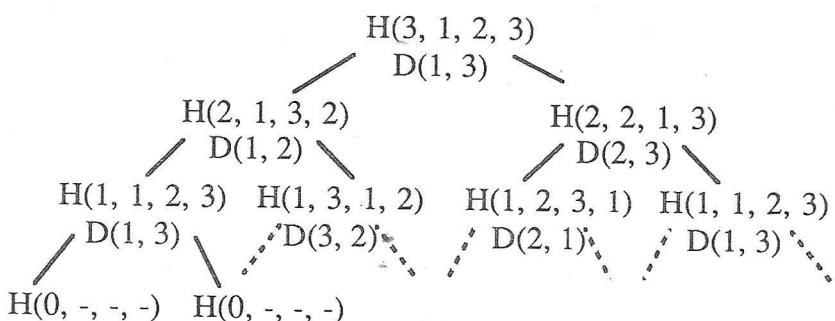


Figure 15. Arbre des appels engendrés par H (3, 1, 2, 3)

Cependant, dans ce cas, le problème est plus simple parce que :

- il est facile de trouver le premier noeud, uniquement en considérant la parité de n, puisque dans la descente à gauche, on inverse les deux derniers paramètres ;
- idem pour le dernier noeud, puisque dans la descente à droite on inverse les deuxième et troisième paramètres; en fait, on n'a pas besoin du dernier ici ;
- en effet, on connaît la longueur de la liste des noeuds à parcourir :  $2^{n-1}$ , donc on sait d'avance le nombre d'itérations ;
- la pile des paramètres (n, a, b, c) est inutile : pour remonter dans l'arbre, et reconstituer les paramètres, il suffit de savoir à chaque étape si l'on remonte une branche gauche ou droite. En fait, une pile de booléens suffit, de hauteur maximum n.

On empile 0 ou 1 en descendant dans l'arbre à gauche ou à droite, et l'on dépile en remontant. On peut voir cette pile comme un **entier** : on multiplie par 2, et on ajoute 1 éventuellement, en descendant ; on divise par 2 en remontant, le reste indiquant si l'on vient de gauche ou de droite.

Le lecteur est invité à écrire l'algorithme dérécurcisé en prenant en compte ces remarques.