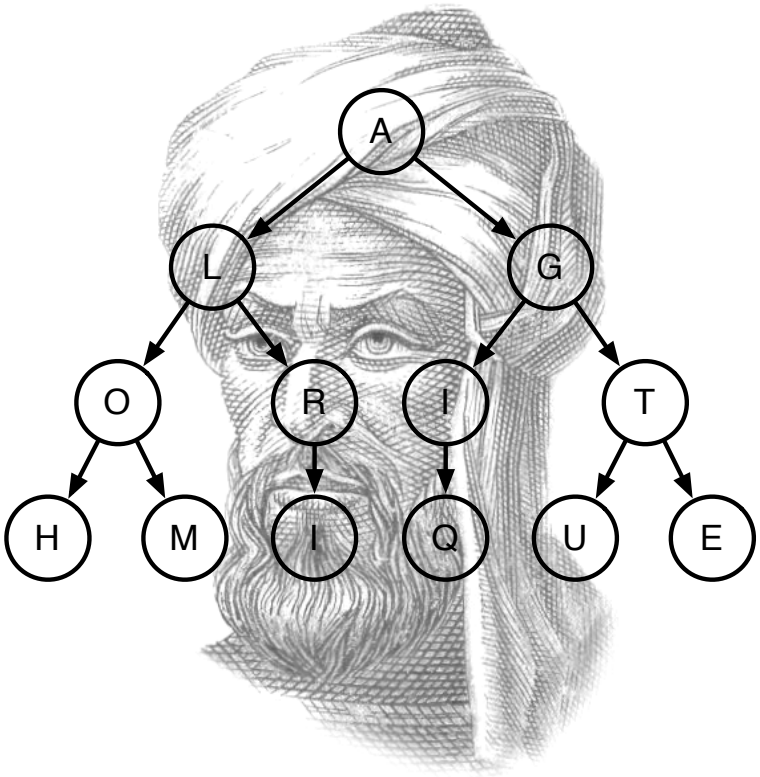
# ALGORITHMIQUE




Notes du cours FS/1/6584
Année préparatoire au master 60 en informatique ULB–UMH

Gilles GEERAERTS (Université Libre de Bruxelles)

ANNÉE ACADÉMIQUE 2008–2009 (2E ÉDITION)

Le personnage en couverture est le mathématicien persan Mohamed IBN MUSSA AL-KHAWARIZMI (783 ?–850 ?) dont le nom a donné le mot français *algorithme*, car il en a décrit plusieurs. Le titre d'un de ses ouvrages a également donné à la langue française le mot *algèbre*. AL-KHAWARIZMI est considéré par beaucoup comme le père de l'algèbre moderne (il a d'ailleurs introduit l'habitude d'appeller $x$ l'inconnue dans une équation). Pour autant, ses travaux ne se limitent pas à ce domaine : on lui doit d'importantes contributions dans les domaines de l'arithmétique, de l'astronomie, ou de la géographie.

Ce document a été mis en page sous LaTeX $2\varepsilon$.

# Table des matières

# Table des figures

# Chapitre 7

# Les arbres

## 7.1 Introduction

Les structures que nous avons considérées jusqu'à présent sont uniquement des structures *linéaires*, dans le sens où :

1. chaque élément de la structure possède au plus *un prédécesseur direct*, et
2. chaque élément de la structure possède au plus *un successeur direct*.

En effet, dans un tableau de taille $n$, chaque case $i$ (pour $1 < i < n$) possède comme unique prédécesseur la case $i-1$, et comme unique successeur la case $i+1$. La case 0 n'a pas de prédécesseur, mais la case 1 est son unique successeur. La case $n$ n'a elle pas de successeur, mais bien un prédécesseur : la case $n-1$. Dans les listes, chaque élément possède un seul et unique champ *next*, qui référence un successeur unique, ou qui contient une valeur conventionnelle pour indiquer qu'il n'y a pas de successeur. Par ailleurs, la tête n'a pas de prédécesseur, et, pour tous les autres éléments $e$, il n'existe qu'un seul autre élément $e'$ dont le champ *next* référence $e$. Enfin, les piles et files sont essentiellement des listes.

Nous allons considérer, dans ce chapitre, une généralisation de ces structures linéaires en levant la restriction numéro 2 : chaque élément sera autorisé à avoir plusieurs successeurs tout en ayant au plus un seul prédécesseur[1]. On a dès lors affaire à une structure qui se *ramifie*, puisque, lors d'un parcours de cette structure, on a, à chaque élément, plusieurs successeurs possibles pour « continuer le parcours ». C'est pour cette raison que ce type de structures est appelé un *arbre*[2].

L'importance des arbres en informatique est *considérable*. D'une part, les structures arborescentes se rencontrent naturellement dans plusieurs cas pratiques de la vie quotidienne que l'on désire modéliser et manipuler à l'aide d'algorithmes. Par exemple l'organisation hiérarchique d'une entreprise est un arbre. La Fig. 7.1 en donne un exemple. D'autre part, les arbres jouent un rôle très important dans de nombreux algorithmes, et ils sont la clef de voûte indispensable pour rendre ces algorithmes efficaces. Sans arbre, pas de base de données, pas de système de fichiers, pas de compression MP3, *etc*.

[1] Nous supposerons également qu'il n'y a pas de *cycle* dans la structure.

[2] Remarquons que si nous relâchons les deux hypothèses ci-dessus, à savoir qu'on autorise également chaque élément à avoir plusieurs prédécesseurs, on obtient un *graphe*. Nous n'aborderons pas les graphes dans ce cours.

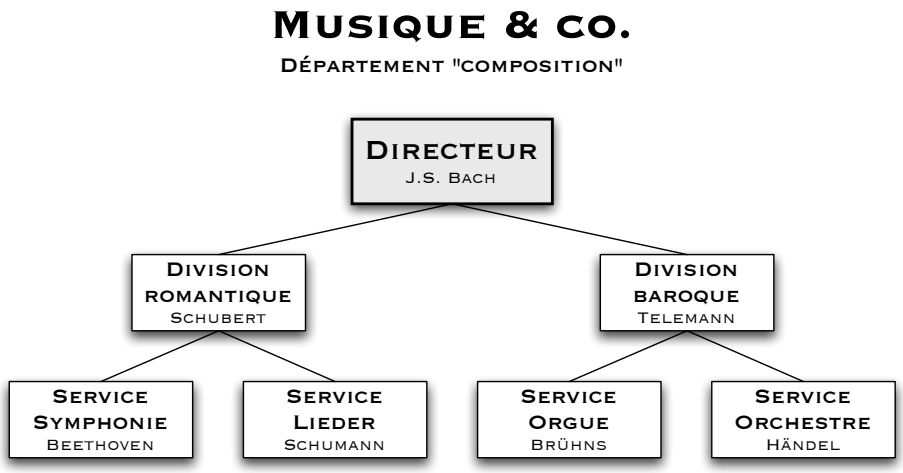



FIG. 7.1 – Un exemple d'arbre, qui représente l'organisation hiérarchique d'une entreprise (un peu fantaisiste).

### 7.1.1 Définitions

Il est possible de donner deux types de définitions des arbres : une définition « classique », en termes d'ensembles ; ou bien une définition récursive. Suivant la définition qu'on adopte, on pourra formuler les algorithmes de façon itérative ou récursive.

Voici d'abord la définition ensembliste :

**Définition 5** *Un arbre est une structure de données composée d'un ensemble N de nœuds. Chaque nœud n est composé d'une information* $\Lambda(n)$ *et d'un ensemble* $\text{Succ}(n)$ *de nœuds successeurs. Par ailleurs, un et un seul nœud de N est appelé la racine, et est dénoté r.*

*Ces éléments respectent les conditions suivantes :*

1. *r n'a pas de prédécesseur : pour tout nœud* $n \in N$, $r \notin \text{Succ}(n)$.
2. *Chaque nœud n'a pas plus d'un prédécesseur : pour tout nœud* $n \in N$, *il n'existe pas de paire de nœuds* $n_1$ *et* $n_2$ *tels que* $n_1 \neq n_2$, $n \in \text{Succ}(n_1)$ *et* $n \in \text{Succ}(n_2)$.
3. *On peut accéder à chaque nœud depuis la racine : pour tout nœud n, il existe une séquence* $n_1 n_2, \ldots, n_k$ *de nœuds telle que :* $n_1 = r$, $n_k = n$ *et, pour tout* $1 \leq i \leq k-1$ *:* $n_{i+1} \in \text{Succ}(n_i)$.

Remarquons que les points 2 et 3 de cette définition obligent chaque nœud (à part la racine) à posséder exactement un prédécesseur. En effet, le point 2 indique que le nombre de prédécesseur de chaque nœud est $\leq 1$, et le point 3 implique que chaque nœud (à part la racine) doit posséder un nombre de prédécesseurs $\geq 1$.

Illustrons cette définition à l'aide d'un exemple :

**Exemple 21** *La Fig. 7.2 présente un exemple d'arbre à 7 nœuds. La relation « a pour successeur » est représentée par une flèche. Nous avons donc* $N = \{n_1, n_2, \ldots, n_7\}$. *La racine est le nœud* $r = n_1$ *: elle est bien dépourvue de prédécesseur. Elle possède 3 successeurs :* $\text{Succ}(n_1) = \{n_2, n_3, n_4\}$. *Certains nœuds n'ont pas de successeur, comme par exemple :* $n_3$ *ou* $n_9$. *Tous les nœuds ont au plus 1 prédécesseur, et sont accessibles à partir de la racine. Par exemple, on accède à* $n_7$, via *la séquence* $n_1 n_4 n_7$.

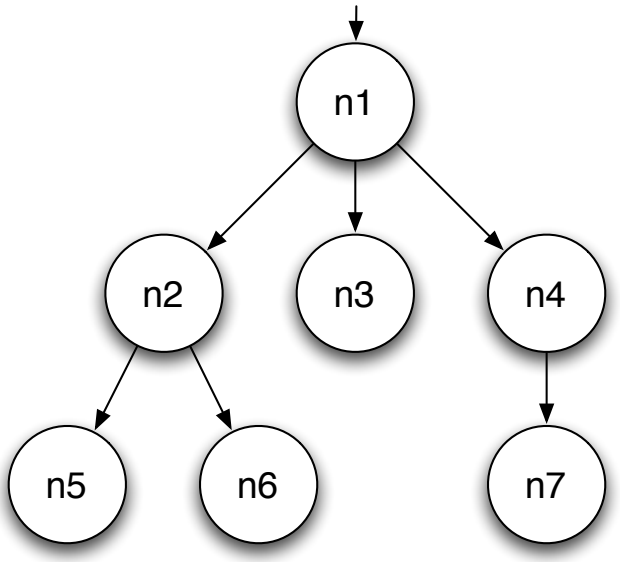



FIG. 7.2 – Un exemple d'arbre. Les informations associées aux nœuds ne sont pas représentées.

Comme on le voit, la définition d'un arbre est plus complexe que celle d'une liste ou d'une autre séquence linéaire. On peut néanmoins obtenir une définition relativement simple si on admet une définition récursive :

**Définition 6** *Un arbre est :*

1. *soit l'*arbre vide
2. *soit un nœud n appelé* racine*, composé d'une information* $\Lambda(n)$ *et d'un ensemble (potentiellement vide)* $\{A_1, A_2, \ldots, A_\ell\}$ *d'arbres non-vides appelés* sous-arbres*.*

Illustrons cette définition :

**Exemple 22** *Reprenons l'arbre de la Fig. 7.2. Nous avons illustré la vue* récursive *de cette arbre à la Fig. 7.3. Cet arbre est maintenant vu de la façon suivante : il est composé du nœud* $n_1$ *(qui constitue sa racine) et de trois sous-arbres A, B et C, représentés en grisé sur la figure. A est un arbre dont la racine est* $n_2$*, et dont les sous-arbres sont D et E. C est un arbre dont la racine est* $n_4$ *et qui ne possède que F comme sous-arbre. B, D, E et F sont des arbres composés de leur seule racine (respectivement* $n_3$*,* $n_5$*,* $n_6$*,* $n_7$*).*

Il n'est pas difficile de se convaincre que les deux définitions sont équivalentes. En effet, l'arbre vide de la Définition 6 correspond à un arbre dont l'ensemble des nœuds est vide. Un nœud $n$ de la Définition 5 correspond à un nœud $n'$ dans la Définition 6, tel que $\Lambda(n') = \Lambda(n)$, et tels que les successeurs de $n'$ sont des arbres $A_1, A_2, \ldots, A_k$ dont les racines $r_1, r_2, \ldots r_k$ constituent l'ensemble $\mathrm{Succ}(n)$.

### 7.1.2 Vocabulaire

Un certain vocabulaire conventionnel est associé aux arbres. Il permet en général de simplifier l'exposé des algorithmes sur les arbres. Nous présentons ce vocabulaire dans

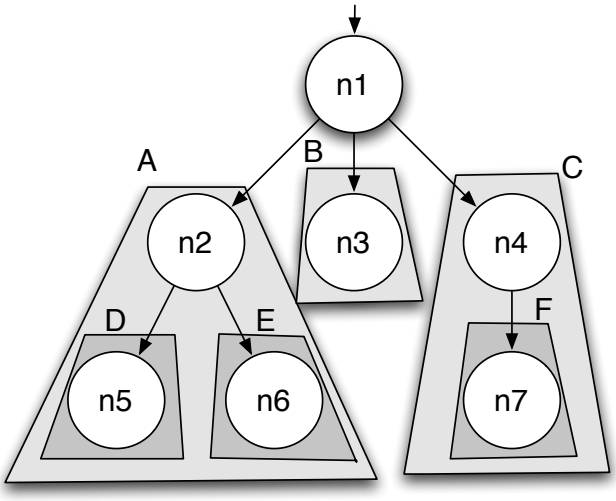



FIG. 7.3 – La vue récursive de l'arbre de la Fig. 7.2.

les cadre de la définition ensembliste, mais il peut aisément être transposé dans le cadre de la vue récursive. Toutes ces définitions sont illustrées par des exemples ci-dessous.

**Fils :** Les nœuds de l'ensemble $\text{Succ}(n)$ sont appelés les *fils* de $n$.

**Père :** Le *père* d'un nœud $n$ est l'unique nœud $n'$ dont $n$ est le fils. Remarquons que la racine n'a pas de père.

**Feuille :** Une *feuille* est un nœud qui n'a pas de fils.

**Nœud interne :** Un nœud est un *nœud interne* si et seulement si il n'est pas une feuille.

**Chemin :** Un *chemin* dans un arbre est une séquence $n_1 n_2 \cdots n_k$ de nœuds de l'arbre telle que, pour tout $1 \leq i < k$ : $n_{i+1} \in \text{Succ}(n_i)$. On dit que ce chemin va de $n_1$ à $n_k$. La *longueur d'un chemin* $n_1 n_2 \cdots n_k$ est le nombre $k$ de nœuds qui le composent.

Remarquons que, d'après les définitions d'un arbre, il n'y a qu'un seul chemin allant de la racine à un nœud $n$ donné.

**Accessible :** Un nœud $n$ est *accessible* à partir d'un nœud $n'$ s'il existe un chemin allant de $n'$ à $n$ dans l'arbre.

**Descendant :** Un nœud $n$ est un *descendant* d'un nœud $n'$ si et seulement si $n$ est accessible à partir de $n'$.

**Hauteur :** La *hauteur d'un arbre* est la longueur du plus long chemin partant de la racine de l'arbre. Dans la suite, la hauteur d'un arbre $A$ est notée $\text{hauteur}(A)$.

**Profondeur :** La *profondeur d'un nœud* $n$ de l'arbre est la longueur du chemin allant de la racine à $n$.

**Exemple 23** *La Fig. 7.4 illustre ce vocabulaire. Elle présente un arbre formé de 12 nœuds, et de hauteur 4, comme en témoigne le chemin en gris :* $n_1 n_2 n_6 n_{11}$. *Un autre chemin dans l'arbre est* $n_4 n_8 n_{12}$ *(flèches en pointillés). Les fils de* $n_4$ *sont* $n_7$, $n_8$ *et* $n_9$ *(et* $n_4$ *est donc le père de ces trois nœuds). Les descendants de* $n_4$ *sont ses fils, plus* $n_{12}$. *La profondeur de* $n_5$ *et* $n_6$ *est 3. Celle de* $n_{10}$ *est 4. Les nœuds* $n_4$ *et* $n_1$ *sont des nœuds internes, alors que* $n_{11}$ *est une feuille.*

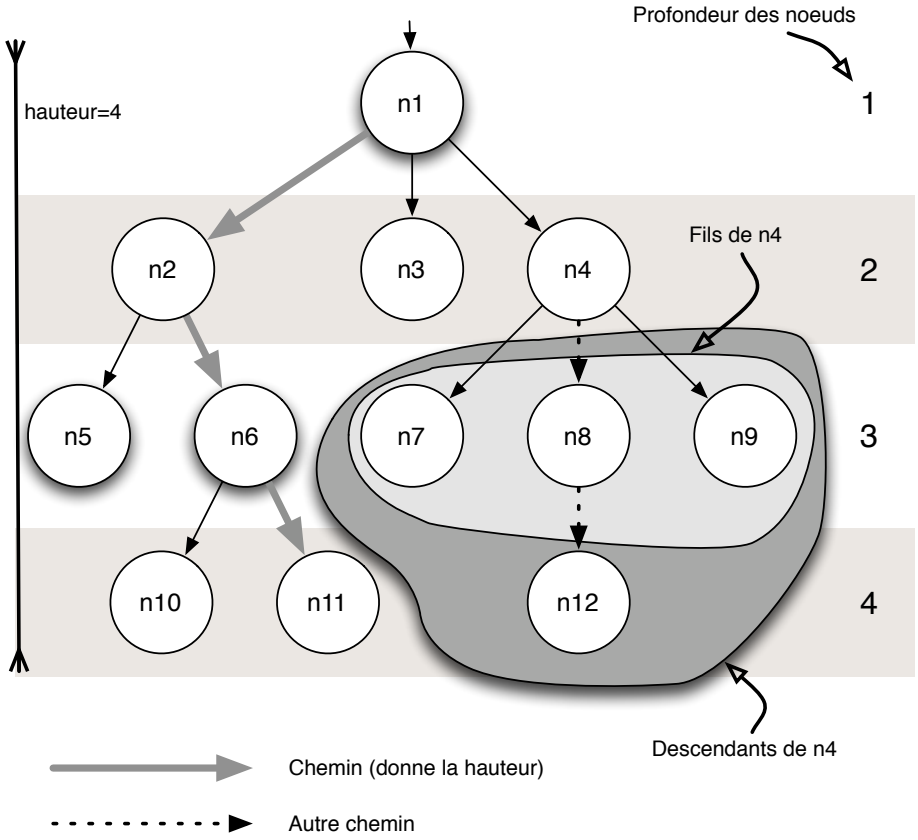



FIG. 7.4 – Quelques illustrations du vocabulaire sur les arbres.

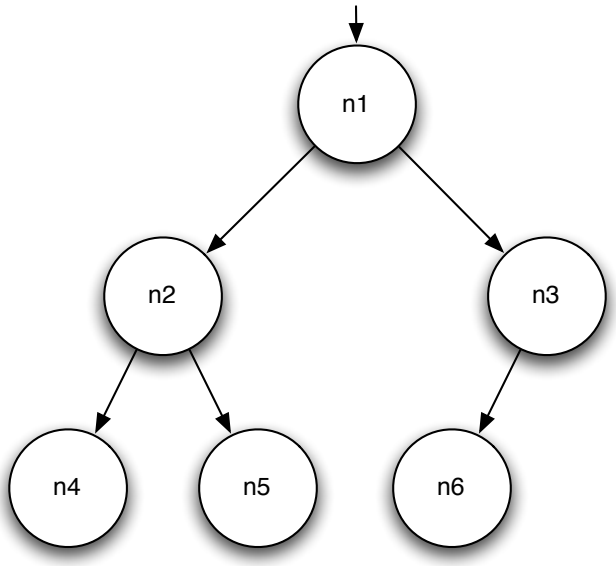



FIG. 7.5 – Un exemple d'arbre binaire équilibré.

### 7.1.3 Cas particuliers

En pratique, les arbres que l'on utilise, ou qui rendent efficaces les algorithmes proposés, sont des *cas particuliers* de la définition générale donnée ci-dessus. Dans cette section, nous détaillons trois restrictions supplémentaires qui seront souvent utilisées par la suite.

**Arbres ordonnés** Dans les définitions données au début de ce chapitre, nous avons toujours considéré qu'un nœud possédait un *ensemble* de fils, ce qui signifie qu'il n'existe pas d'ordre sur ces fils. En pratique, on fixera souvent un ordre, et on parlera alors du *premier fils*, du *second fils*, du *dernier fils*, etc.

**Arbres *n*-aires** Pour différentes raisons, il sera en général pratique de *limiter* le nombre de fils qu'un nœud peut posséder. On parle alors d'arbre *n*-aire :

**Définition 7** *Un arbre n-aire est un arbre dont chaque nœud possède au plus n fils.*

Le cas le plus utilisé en pratique (et dans la suite de ce cours), et l'*arbre binaire*, où chaque nœud possède au plus $n = 2$ fils. Dans la suite, nous supposerons implicitement qu'un arbre binaire est ordonné, et nous appellerons respectivement *fils gauche* et *fils droit* le premier et le second fils d'un nœud. Dans le cas des arbres binaires (ordonnés) nous autoriserons un nœud à posséder un fils droit mais pas de fils gauche. (ce qui revient à supposer que le fils gauche est vide).

**Exemple 24** *La Fig. 7.5 présente un exemple d'arbre binaire. En effet, les nœuds $n_1$ et $n_2$ ont deux fils ; le nœuds $n_3$ a un fils ; et les nœuds $n_4$, $n_5$ et $n_6$ n'ont pas de fils. $n_2$ est le* fils gauche *de $n_1$ et $n_3$ est le* fils droit *de $n_1$.*

**Arbres équilibrés** Comme nous le verrons dans la suite, la complexité de plusieurs algorithmes sur les arbres dépend de la *hauteur* des arbres auxquels ils sont appliqués. Dans ce contexte, les arbres dont les nœuds sont répartis entre les différents sous-arbres de « façon équitable » seront les plus favorables (du point de vue du temps d'exécution).

La définition d'*arbre équilibré* (*balanced tree*) capture cette notion de « répartition équitable des nœuds dans les sous-arbres ». Intuitivement, pour qu'un arbre composé d'un nœud $n$ et de sous-arbres $A_1, A_2, \ldots, A_k$ soit équilibré, il faut qu'il respecte deux critères. Tout d'abord, il faut que chacun de ses sous-arbres soit équilibré. Ensuite, il faut que les hauteurs des sous-arbres soit *à peu près* équivalentes. Plus rigoureusement, nous demanderons que les hauteurs de deux sous-arbres $A_1$ et $A_2$, quels qu'ils soient, différent au plus de 1. En effet, nous ne pouvons pas imposer que tous les sous-arbres aient la même hauteur, car il se pourrait qu'il n'y ait pas assez de nœuds pour satisfaire ce critère. Par exemple, un arbre binaire à deux nœuds $n_1$ et $n_2$, tel que $n_1$ est la racine, et $n_2$ le fils gauche de $n_1$ doit être considéré comme *équilibré*, même si le sous-arbre gauche de $n_1$ contient plus de nœuds (il en contient 1) que son sous-arbre droit (qui n'en contient pas). Autrement, il ne serait pas possible de construire d'arbre binaire équilibré avec deux nœuds (ni avec 4, 5, 6,... nœuds).

Nous pouvons maintenant donner la définition d'arbre équilibré. Elle est formulée de façon récursive :

**Définition 8** *L'arbre vide est* équilibré. *Un arbre A, non-vide, est* équilibré *si et seulement si :*

1. *chacun de ses sous-arbre est équilibré et*
2. *pour toute paire $A_1$, $A_2$ de sous-arbres de A,* $|\text{hauteur}(A_1) - \text{hauteur}(A_2)| \leq 1$

**Exemple 25** *L'arbre de la Fig. 7.5 est un* arbre binaire équilibré.

**Propriété des arbres binaires équilibrés** Montrons maintenant que la définition d'*arbre équilibré* que nous venons de donner satisfait bien notre critère de « hauteur minimale ». Pour ce faire, nous allons prouver qu'un arbre équilibré contenant $n$ nœuds a une hauteur $h = \mathcal{O}(\log(n))$, ce qui, intuitivement, signifie que la hauteur est « petite » par rapport au nombre de nœuds.

Cette preuve requiert quelques pré-requis mathématiques que nous allons maintenant exposer.

Tout d'abord, nous devons définir les *nombres de Fibonacci*[3]. Il s'agit d'une séquence infinie de nombres appelés $F_1, F_2, F_3, \ldots$, telle que chaque nombre est égal à la somme des deux précédents (en commençant avec $F_1 = 1$ et $F_2 = 1$). Voici la définition (inductive) formelle :

**Définition 9** *La séquence $F_1, F_2, F_3, \ldots$ des* nombres de Fibonacci *est une séquence de nombre entiers définie inductivement comme suit :*

- $F_1 = 1$ *;*
- $F_2 = 1$ *;*
- *pour tout* $i \geq 3$ *:* $F_i = F_{i-1} + F_{i-2}$.

[3]D'après le nom du mathématicien italien Leonardo FIBONACCI (c. 1175 - c. 1250). Son *Liber Abaci* contribua à introduire les chiffres arabes en occident. C'est dans ce livre qu'on trouve un exercice consistant à calculer la séquence qui porte son nom. Néanmoins, cette séquence était déjà connue des mathématiciens indiens, bien avant lui.

Cette séquence est donc :

$$1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,\ldots$$

Il a été établi que cette suite a plusieurs connexions avec le nombre d'or[4] $\varphi$ qui vaut :

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1,618\ldots$$

Plus précisément, on sait que :

$$\forall i \geq 1 \quad : \quad F_i \geq \frac{\varphi^i}{\sqrt{5}} - 1 \tag{7.1}$$

Tous ces résultats peuvent être trouvés dans des ouvrages classiques de mathématiques discrètes, comme [**?**]. Nous pouvons maintenant prouver notre théorème :

**Théorème 5** *Soit A un arbre binaire équilibré composé de n nœuds, et de hauteur h. Alors,* $h = \mathcal{O}(\log(n))$.

*Preuve.* Nous allons tout d'abord prouver que si $A$ est un arbre de $n$ nœuds et de hauteur $h$, alors $F_h \leq n$, c'est-à-dire qu'un tel arbre contient nécessairement un nombre de nœuds qui est supérieur au $h^{\text{e}}$ nombre de Fibonacci. Nous allons établir ce résultat par induction sur $h$.

**Cas de base** $h = 1$ ou $h = 2$. Dans le cas où $h = 1$, l'arbre possède exactement $n = 1$ nœud. Nous avons bien $F_1 = 1$. Dans le cas où $h = 2$, l'arbre possède au moins 2 nœuds. Nous avons donc $n \geq 2$. Or $2 \geq F_2 = 1$.

**Cas inductif** $h = \ell > 2$. L'*hypothèse d'induction est* : tout arbre binaire équilibré de hauteur $h' \leq \ell - 1$ possède au moins $F_{h'}$ nœuds. D'après la définition d'un arbre équilibré, les deux sous-arbres de la racine sont équilibrés. D'après cette définition toujours, nous avons deux possibilités. Soit les deux fils sont de hauteur $\ell - 1$. Soit l'un des deux est de hauteur $\ell - 1$ et l'autre de hauteur $\ell - 2$.

Considérons d'abord le second cas. Le fils qui est de hauteur $\ell - 1$ (et est aussi équilibré) possède, par hypothèse d'induction, au moins $n_1 \geq F_{\ell-1}$ nœuds. L'autre fils, possède, pour la même raison, au moins $n_2 \geq F_{\ell-2}$ nœuds. Or, l'arbre de hauteur $\ell$ que nous considérons possède $n_1 + n_2 + 1$ nœuds. Donc :

$$\begin{aligned} n &= n_1 + n_2 + 1 \\ &> n_1 + n_2 \\ &\geq F_{\ell-1} + F_{\ell-2} \\ &= F_\ell \\ &= F_h \end{aligned}$$

Nous avons donc bien $F_h \leq n$.

Dans le cas où chaque fils est de hauteur $\ell - 1$, il est facile de voir que l'arbre résultant contiendra au moins autant de nœuds que dans le cas où l'un des fils est de hauteur $\ell - 1$ et l'autre de hauteur $\ell - 2$. Nous avons donc aussi $F_h \leq n$.

[4] Le nombre d'or est une valeur qui était régulièrement utilisée dans les mesures des statues ou des temples grecs. Cette proportion était considérée par les Grecs comme particulièrement harmonieuse. Le nom $\varphi$ est choisi en hommage au sculpteur Φειδίας (Phidias).

Nous savons donc maintenant que pour tout arbre binaire équilibré possédant $n$ nœuds et de hauteur $h$ : $n \geq F_h$. Or, grâce à l'équation (7.1), nous obtenons :

$$\begin{aligned} & n \geq F_h \geq \frac{\varphi^h}{\sqrt{5}} - 1 \\ \Rightarrow \quad & n \geq \frac{\varphi^h}{\sqrt{5}} - 1 \\ \Rightarrow \quad & \sqrt{5}(n+1) \geq \varphi^h \\ \Rightarrow \quad & \log_\varphi \left(\sqrt{5}(n+1)\right) \geq h \\ \Rightarrow \quad & \log_\varphi \left(\sqrt{5}\right) + \log_\varphi (n+1) \geq h \end{aligned}$$

Nous voyons donc que $\log_\varphi \left(\sqrt{5}\right) + \log_\varphi (n+1)$ est une *borne supérieure* de la hauteur $h$. Nous pouvons donc utiliser la notation $\mathcal{O}$, dans laquelle $\log_\varphi \left(\sqrt{5}\right)$ disparaît, puisqu'il s'agit d'une constante, et nous obtenons : $h = \mathcal{O}\left(\log_\varphi (n+1)\right)$, ce qui est en $\mathcal{O}(\log(n))$ □

### 7.1.4 Implémentation

Nous pouvons maintenant nous tourner vers des considérations plus pratiques, à savoir l'implémentation de la structure d'arbre. L'implémentation que nous allons présenter ici utilise des pointeurs et est complètement dynamique. Remarquons qu'il est également possible d'implémenter un arbre dans un tableau, mais nous ne discuterons pas cette possibilité ici.

De manière générale, un arbre est composé de nœuds qui possèdent chacun un certain nombre de *successeurs*, que nous avons appelés *fils*. La structure de base sera donc celle qui permet d'implémenter un nœud, et un arbre sera essentiellement un pointeur vers le nœud racine (ou bien **NIL** si l'arbre est vide).

Comme, de manière générale, le nombre de *fils* d'un nœud n'est pas borné, il faut prévoir, dans la structure qui représente un nœud, une *liste* contenant les adresses de tous les fils du nœud. On obtient alors les déclarations suivantes. Tout d'abord, les éléments de la liste de successeurs :

```
struct ElemArbre
    Nœud * info ;
    ElemArbre * next ;
```

Puis, les nœuds à proprement parler :

```
struct Nœud
    Entier info ;
    ElemArbre * succ ;
```

En pratique, la gestion d'une *liste* pour stocker les adresses des fils d'un nœud est souvent considérée comme lourde, surtout si on a affaire à un arbre ordonné et que l'on désire pouvoir accéder *immédiatement* au $k^{\text{e}}$ fils d'un nœud. C'est pourquoi on adopte souvent l'hypothèse simplificatrice qui consiste à considérer qu'on ne manipule que des arbres $N$-aires, pour un $N$ « suffisamment grand ». On peut dès lors stocker les adresses des fils dans un *tableau* de successeurs, les cases inutilisées étant initialisées à **NIL** :

```
struct Nœud
    Entier info ;
    Nœud * succ[N] ;
```

Finalement, dans le cas des *arbres binaires* – de loin les arbres les plus courants – on peut explicitement différencier *fils gauche* et *fils droit* dans deux champs du nœud :

```
struct Nœud
    Entier info ;
    Nœud * fg ;
    Nœud * fd ;
```

À titre d'exemple, nous présentons maintenant quelques fonctions qui manipulent la structure d'arbre binaire. Elles sont données à l'Algorithme 30 et à l'Algorithme 31, et commentées ci-dessous :

**ArbreVide** Cette fonction renvoie un arbre vide, c'est-à-dire le pointeur **NIL**.

**CréerRacine** Cette fonction renvoie un pointeur vers une racine (un arbre ne contenant qu'un seul nœud), étiqueté par *i*.

**InsèreGauche et InsèreDroite** Ces deux fonctions insèrent respectivement un nouveau nœud portant l'information *i* comme fils gauche (droit) de l'arbre *A*. On suppose donc que l'arbre *A* n'a pas encore de fils gauche (droit).

**EstVide** Cette fonction renvoie vrai si et seulement si l'arbre *A* est vide (c'est-à-dire ssi son pointeur racine est égal à **NIL**).

**EstFeuille** Cette fonction renvoie vrai si et seulement si l'arbre passé est une feuille (c'est-à-dire ssi ses deux fils sont **NIL**).

**FilsGauche et FilsDroit** Ces fonctions renvoient respectivement un pointeur vers le fils gauche (droit) de l'arbre *A*.

**InfoRacine** Cette fonction permet de consulter l'information stockée dans la racine de l'arbre.

Comme on peut le voir, les fonctions d'insertions données ne fonctionnent que dans le cas où l'on insère une nouvelle feuille. De plus, nous n'avons pas donné de fonction de suppression. Ces restrictions sont justifiées par le fait qu'il existe plusieurs manières d'insérer ou de supprimer dans un arbre binaire, qui dépendent de l'information stockée dans l'arbre, et de la façon dont cette information est stockée. Nous étudierons, dans le chapitre suivant, un cas particulier, appelé *arbre binaire de recherche*, pour lequel nous fournirons les fonctions d'insertion et de suppression adaptées.

### 7.1.5 Application : les arbres d'expressions

Une application classique des arbres binaires consiste à représenter et manipuler des expressions arithmétiques. Dans cette section, nous considérerons des expressions formées :

- de nombres entiers
- d'opérateurs $+$, $-$, $*$ ou $/$
- de parenthèses

Comme les opérateurs considérés sont tous *binaires*, la structure d'arbre binaire convient particulièrement bien à leur représentation. En effet, une expression comme $3+5$, par exemple, consiste à appliquer l'opérateur $+$ sur les valeurs 3 et 5, ce qui peut se représenter par un arbre dont la racine est étiquetée par $+$, et les fils gauche et droit par 3 et 5 respectivement. Si nous considérons maintenant l'expression $(3+5)*2$,

```
Nœud * ArbreVide()
début
  | retourner NIL ;
fin

Nœud * CréerRacine( Entier i)
début
  | Nœud * A := new Nœud ;
  | A.info := i ;
  | A.fg := NIL ;
  | A.fd := NIL ;
  | retourner A ;
fin

InsèreGauche(Nœud * A, Entier i)
début
  | A.fg := new Nœud ;
  | A.fg.info := i;
  | A.fg.fg := NIL;
  | A.fg.fd := NIL;
fin

InsèreDroite(Nœud * A, Entier i)
début
  | A.fd := new Nœud ;
  | A.fd.info := i;
  | A.fd.fg := NIL;
  | A.fd.fd := NIL;
fin
```

**Algorithme 30** : Quelques fonctions de base pour manipuler un arbre binaire stockant des entiers.

```
Booléen EstVide(Nœud * A)
début
    si A = NIL alors
        retourner vrai ;
    sinon
        retourner faux ;
fin

Booléen EstFeuille(Nœud * A)
début
    si EstVide(FilsGauche(A)) et EstVide(FilsDroit(A)) alors
        retourner vrai ;
    sinon
        retourner faux ;
fin

Nœud * FilsGauche(Nœud * A)
début
    retourner A.fg ;
fin

Nœud * FilsDroit(Nœud * A)
début
    retourner A.fd ;
fin

Entier InfoRacine(Nœud * A)
début
    retourner A.info ;
fin
```

**Algorithme 31** : Quelques fonctions de base pour manipuler un arbre binaire stockant des entiers (suite)

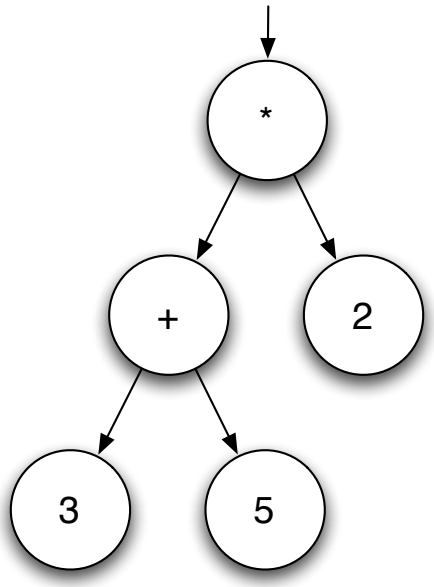



FIG. 7.6 – Un exemple d'arbre d'expression qui représente $(3+5)*2$.

elle consiste en la multiplication de l'expression $3+5$ par l'expression 2. Nous pouvons donc construire une arbre dont la racine est étiquetée par $*$, dont le fils gauche est l'arbre correspondant à $3+5$, et le fils droit est étiqueté par 2, comme illustré à la Fig. 7.6. Comme on le voit, la priorité fixée par les parenthèses est implicite dans l'arbre : afin de pouvoir évaluer le $*$, on devra d'abord évaluer le $+$, puisque celui-ci est son fils gauche.

Nous pouvons maintenant définir un *arbre d'expression* :

**Définition 10** *Un arbre d'expression est un arbre binaire, tel que :*
- *Chaque feuille est étiquetée par un nombre entier i, et représente l'expression i ;*
- *Chaque nœud interne :*
  - *est étiqueté par un opérateur op ;*
  - *possède un fils gauche $fg$* et *un fils droit $fd$ ;*
  - *représente l'expression $(E_1)$ op $(E_2)$, où $E_1$ est l'expression représentée par $fg$, et $E_2$ est l'expression représentée par $fd$.*

Nous verrons, dans la section suivante, que les algorithmes généraux de parcours permettent de manipuler de façon très élégante ces arbres d'expression.

## 7.2 Parcours des arbres binaires

Supposons que nous désirions appliquer un certain traitement à chaque nœud d'un arbre binaire $A$ (par exemple : afficher l'information contenue dans la racine). Pour ce faire, il nous faudra certainement : traiter la racine de l'arbre, traiter le sous-arbre gauche et traiter le sous-arbre droit. S'il est raisonnable de dire que le sous-arbre gauche doit être traité *avant* le sous-arbre droit (sinon, pourquoi aurait-on choisi cet ordre ?),

il est plus difficile de décider *à quel moment* il y a lieu de traiter la racine. Est-ce avant d'avoir traité les deux fils ? après les avoir traités ? entre les deux traitements ?

Contrairement à une liste, un arbre ne possède pas *un seul ordre naturel* sur ses éléments, mais bien plusieurs. Ces différents ordres sont formalisés par la notion de *parcours*, c'est-à-dire de procédure traversant tous les nœuds de l'arbre et les traitant chacun une et une seule fois à un moment précis. L'ordre dans lequel les nœuds sont traités fixe l'ordre sur les nœuds.

Dans cette section, nous étudions quatre parcours. Les trois premiers suivent le canevas que nous venons d'exposer (traitement de la racine, du fils gauche et du fils droit), en faisant varier le moment auquel la racine est traitée. Ces parcours sont très faciles à exprimer de façon récursive, mais nous en étudierons également des versions itératives, qui utilisent une pile. Nous étudierons ensuite le parcours par niveaux, qui fait appel à la structure de file du Chapitre 6. Ces parcours sont illustrés à la Fig. 7.7 et sont donnés aux Algorithmes 32 à 37.

```
ParcoursPréfixé(Nœud * A)
début
    si ¬EstVide(A) alors
        Traiter A ;
        ParcoursPréfixé(FilsGauche(A)) ;
        ParcoursPréfixé(FilsDroit(A)) ;
fin

ParcoursInfixe(Nœud * A)
début
    si ¬EstVide(A) alors
        ParcoursInfixe(FilsGauche(A)) ;
        Traiter A ;
        ParcoursInfixe(FilsDroit(A)) ;
fin

ParcoursPostfixé(Nœud * A)
début
    si ¬EstVide(A) alors
        ParcoursPostfixé(FilsGauche(A)) ;
        ParcoursPostfixé(FilsDroit(A)) ;
        Traiter A ;
fin
```

**Algorithme 32** : Le trois parcours « de base » exprimés de façon récursive.

### 7.2.1 Parcours préfixé ou parcours en profondeur

Comme son nom le suggère, le parcours *préfixé* consiste à traiter la racine de l'arbre *avant* de traiter le fils gauche et le fils droit. Ce traitement n'a pas lieu si l'arbre passé

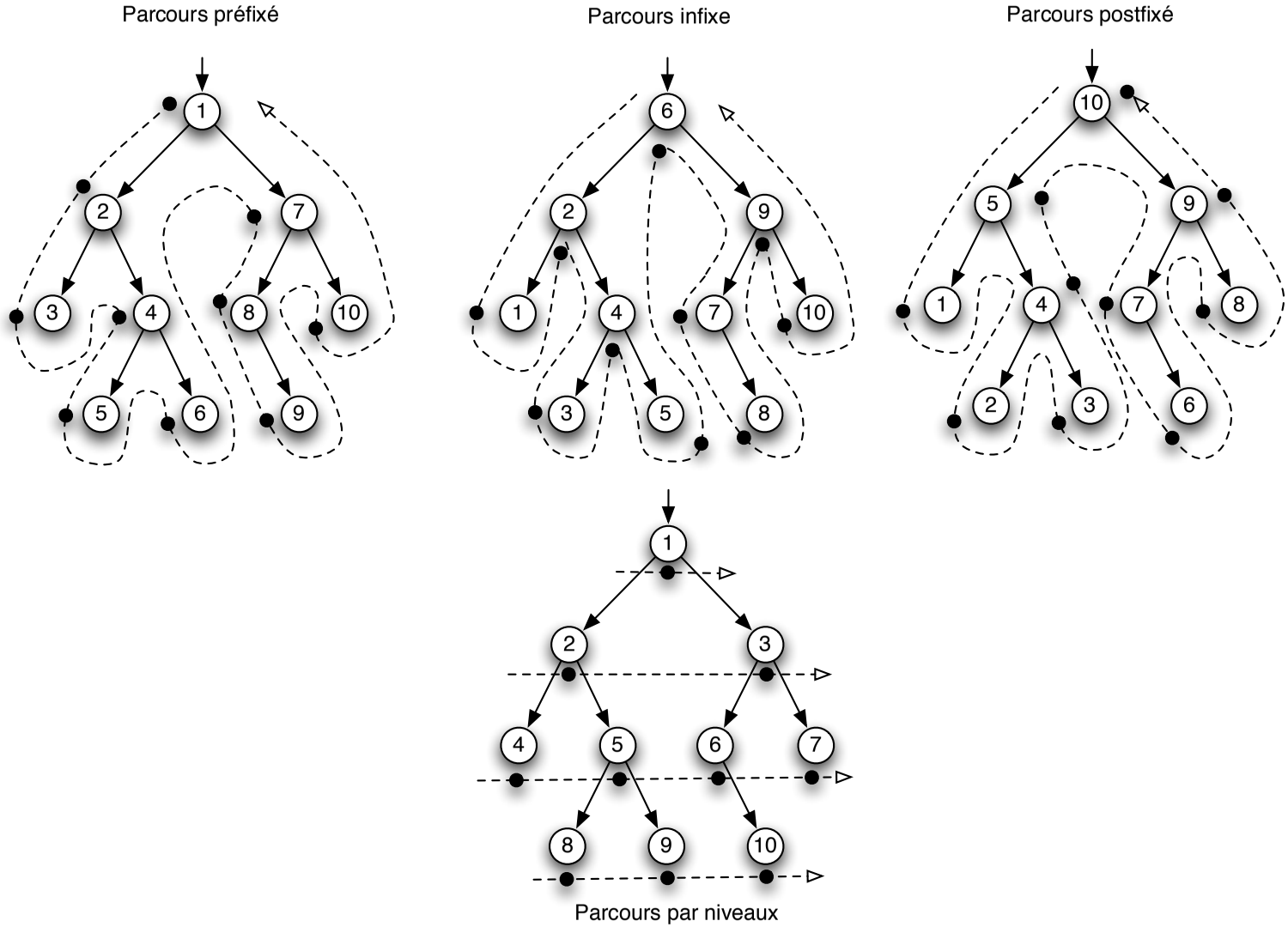



FIG. 7.7 – Les quatre parcours d'arbre classiques. La flèche en pontillé indique l'ordre des appels. Un point à proximité d'un nœud indique à quel moment le nœud sera effectivement traité lors du parcours.

```
ParcoursPréfixé(Nœud * A)
début
    Pile P ;
    P :=PileVide() ;
    P :=Push(P, (A, d)) ;
    tant que ¬EstVide(P) faire
        (A, action) := Top(P) ;
        P := Pop(P) ;
        si ¬EstVide(A) alors
            si action = d alors
                P :=Push(P, (FilsDroit(A), d)) ;
                P :=Push(P, (FilsGauche(A), d)) ;
                P :=Push(P, (A, t)) ;
            sinon
                Traiter A ;
fin
```

**Algorithme 33** : Le parcours préfixé exprimé de façon itérative.

```
ParcoursInfixe(Nœud * A)
début
    Pile P ;
    P :=PileVide() ;
    P :=Push(P, (A, d)) ;
    tant que ¬EstVide(P) faire
        (A, action) := Top(P) ;
        P := Pop(P) ;
        si ¬EstVide(A) alors
            si action = d alors
                P :=Push(P, (FilsDroit(A), d)) ;
                P :=Push(P, (A, t)) ;
                P :=Push(P, (FilsGauche(A), d)) ;
            sinon
                Traiter A ;
fin
```

**Algorithme 34** : Le parcours infixe exprimé de façon itérative.

```
ParcoursPostfixé(Nœud * A)
début
    Pile P ;
    P :=PileVide() ;
    P :=Push(P, (A, d)) ;
    tant que ¬EstVide(P) faire
        (A, action) := Top(P) ;
        P := Pop(P) ;
        si ¬EstVide(A) alors
            si action = d alors
                P :=Push(P, (A, t)) ;
                P :=Push(P, (FilsDroit(A), d)) ;
                P :=Push(P, (FilsGauche(A), d)) ;
            sinon
                Traiter A ;
fin
```

**Algorithme 35** : Le parcours infixe exprimé de façon itérative.

à la fonction est vide. La formulation récursive de ce parcours est donc extrêmement simple, et est donnée à l'Algorithme 32.

Intéressons-nous maintenant à une implémentation *itérative* de ce parcours. Pour ce faire, nous devons d'abord mieux comprendre *comment* il fonctionne. Considérons l'exemple de la Fig. 7.5. Sur cet arbre, le parcours préfixé traite les nœuds dans l'ordre suivant : $n_1, n_2, n_4, n_5, n_3, n_6$. Initialement, l'appel récursif est effectué sur $n_1$ : $n_1$ est traité, puis on effectue l'appel récursif sur $n_2$. À ce moment, l'appel récursif sur $n_3$ n'est pas encore exécuté, et on peut donc considérer qu'il est « en attente ». L'appel récursif sur $n_2$ effectue le traitement de $n_2$, ainsi qu'un nouvel appel sur $n_4$, ce qui met l'appel récursif sur $n_5$ « en attente », mais *avant* l'appel sur $n_3$ lui aussi « en attente ».

Une façon de résumer cette explication, consiste à exprimer le travail effectué par l'algorithme sous forme d'une *séquence d'actions*, qui peuvent être de deux types :

1. soit l'algorithme doit effectuer l'appel récursif sur un nœud, ce que nous appellerons *développer le nœud*. L'action consistant à développer l'arbre dont la racine est $n$ sera indiquée dans la séquence par $(n, \mathtt{d})$.
2. soit l'algorithme doit *traiter le nœud*. L'action consistant à traiter l'arbre dont la racine est $n$ sera indiquée dans la séquence par $(n, \mathtt{t})$.

Les actions présentes dans la séquence sont exécutées l'une après l'autre par l'algorithme : la première est donc l'action *courante* et les suivantes sont *en attente*. Exécuter l'action consiste parfois à remplacer cette action par plusieurs autres actions dans la séquence : dans le cas du développement de $A$, on doit insérer les actions qui indiquent qu'il faut traiter $A$ et développer ses deux fils.

Initialement, la seule action qui doit être effectuée est le *développement de* $n_1$, ce que nous noterons par la séquence suivante : $(n_1, \mathtt{d})$. Développer $n_1$ consiste tout d'abord à traiter $n_1$, puis à développer le fils gauche de $n_1$, puis à développer le fils droit de $n_1$. Nous pouvons donc remplacer $(n_1, \mathtt{d})$ par $(n_1, \mathtt{t}), (n_2, \mathtt{d}), (n_3, \mathtt{d})$. L'algorithme doit donc maintenant traiter $n_1$, comme indiqué par le premier élément de la séquence. Une fois que cela est fait, nous pouvons supprimer $(n_1, \mathtt{t})$ de la séquence des actions à effectuer, et nous voyons qu'il nous reste à effectuer $(n_2, \mathtt{d}), (n_3, \mathtt{d})$. Développer $n_2$ revient à traiter $n_2$, puis développer $n_4$ et $n_5$. Nous obtenons donc la séquence

$(n_2, \mathtt{t}), (n_4, \mathtt{d}), (n_5, \mathtt{d}), (n_3, \mathtt{d})$. On voit que le développement de $n_3$ reste « en attente », et que ce développement aura lieu avant les développements de $n_4$ et $n_5$. L'algorithme continue ainsi et s'arrête lorsque la séquence est vide :

$$
\begin{array}{ll}
 & (n_1, \mathtt{d}) \\
\rightarrow & (n_1, \mathtt{t}), (n_2, \mathtt{d}), (n_3, \mathtt{d}) \\
\rightarrow & (n_2, \mathtt{d}), (n_3, \mathtt{d}) \\
\rightarrow & (n_2, \mathtt{t}), (n_4, \mathtt{d}), (n_5, \mathtt{d}), (n_3, \mathtt{d}) \\
\rightarrow & (n_4, \mathtt{d}), (n_5, \mathtt{d}), (n_3, \mathtt{d}) \\
\rightarrow & (n_4, \mathtt{t}), (\mathbf{NIL}, \mathtt{d}), (\mathbf{NIL}, \mathtt{d}), (n_5, \mathtt{d}), (n_3, \mathtt{d}) \\
\rightarrow & (n_5, \mathtt{d}), (n_3, \mathtt{d}) \\
\rightarrow & (n_5, \mathtt{t}), (\mathbf{NIL}, \mathtt{d}), (\mathbf{NIL}, \mathtt{d}), (n_3, \mathtt{d}) \\
\rightarrow & (n_3, \mathtt{d}) \\
\rightarrow & (n_3, \mathtt{t}), (n_6, \mathtt{d}), (\mathbf{NIL}, \mathtt{d}) \\
\rightarrow & (n_6, \mathtt{d}), (\mathbf{NIL}, \mathtt{d}) \\
\rightarrow & (n_6, \mathtt{t}), (\mathbf{NIL}, \mathtt{d}), (\mathbf{NIL}, \mathtt{d}) \\
\rightarrow & \varepsilon
\end{array}
$$

Il est facile de voir que les manipulations effectuées sur la séquence suivent une politique de pile, dont le sommet est le début de la séquence. On obtient ainsi un algorithme *itératif* qui effectue le parcours préfixé d'un arbre $A$. Cet algorithme maintient, à tout moment, la séquence décrite ci-dessus dans une pile, avec le début de la séquence au sommet. Initialement, la pile contient $(A, \mathtt{d})$. Ensuite, tant que la pile n'est pas vide, l'algorithme effectue les actions suivantes. Il commence par enlever un couple $(A, act)$ du sommet de la pile, indiquant que l'action $act$ doit être appliquée à l'arbre $A$ :

1. si l'arbre $A$ est vide, rien n'est effectué ;
2. si l'action est `t`, l'algorithme traite $A$ ;
3. si l'action est `d`, l'algorithme effectue le **Push** de : $(\mathbf{FilsDroit}(A), \mathtt{d})$, puis de $(\mathbf{FilsGauche}(A), \mathtt{d})$ et enfin de $(A, \mathtt{t})$. Remarquons que l'ordre est « inversé » par rapport à la spécification naturelle du parcours préfixé. C'est dû au fait qu'on utilise une pile, et que la prochaine action effectuée sera dès lors la dernière insérée dans la pile.

Cet algorithme est donné à l'Algorithme 33.

L'algorithme que nous venons de présenter a l'avantage d'être aisé à expliquer, et d'offrir un canevas général d'algorithme de parcours, comme nous le verrons ci-dessous. Néanmoins, il est possible d'en obtenir une version plus simple, grâce aux deux constatations suivantes :

1. Le *développement* d'un arbre $A$ consiste à effectuer les **Push** successifs de ses sous-arbres droit et gauche (avec l'action `d`), puis de lui-même, accompagné de l'action `t`. Or, comme $(A, \mathtt{t})$ est la dernière information mise sur la pile, c'est aussi la première qui en sortira. Autrement dit, chaque fois que l'on met $(A, \mathtt{t})$, on a la garantie que la prochaine chose à effectuer sera de traiter $A$. Dans ce cas, il n'est pas nécessaire de mettre $(A, \mathtt{t})$ sur la pile : il suffit de traiter $A$ avant de stocker ses deux fils. Comme on ne met désormais plus d'élément de la forme $(A, \mathtt{t})$, il n'est plus nécessaire de spécifier l'action `d` pour ceux qu'on y stocke.
2. Nous pouvons à nouveau appliquer le même raisonnement. Après avoir traité $A$, nous stockons $\mathbf{FilsGauche}(A)$ et $\mathbf{FilsDroit}(A)$ sur la pile. Cela signifie que la prochaine action à effectuer consiste à développer le fils gauche, c'est-à-dire,

stocker **FilsGauche**(**FilsGauche**($A$)) et **FilsDroit**(**FilsGauche**($A$)) sur la pile, *etc*. On voit qu'il n'est pas nécessaire de faire transiter le fils gauche sur la pile si c'est pour l'en retirer immédiatement après.

Au final, l'algorithme fonctionne comme suit. Recevant un arbre $A$, il traite sa racine, **Push** son sous-arbre droit sur la pile, et « descend » dans son fils gauche grâce à $A :=$ **FilsGauche**($A$). Ce traitement est recommencé jusqu'à ce qu'il n'y ait plus de fils gauche. À ce moment, on **Pop** le dernier fils droit en attente de la pile, et on recommence le même traitement. On s'arrête dès que la pile est vide. L'Algorithme 36 présente cette solution.

```
ParcoursPréfixé(Nœud * A)
début
    si ¬EstVide(A) alors
        Pile P ;
        P :=PileVide() ;
        répéter
            Traiter A ;
            P :=Push(P, FilsDroit(A)) ;
            A := FilsGauche(A) ;
            si EstVide(A) et ¬EstVide(P) alors
                A := Top(P) ;
                P := Pop(P) ;
        jusqu'à EstVide(P) ;
fin
```

**Algorithme 36** : Une version simplifiée du parcours préfixé.

### 7.2.2 Parcours infixe

Dans le parcours *infixe*, on traite la racine *après* avoir traité le fils gauche, mais *avant* d'avoir traité le fils droit. La version récursive est donnée à l'Algorithme 32.

Il est facile de voir – comme indiqué sur la Fig. 7.7 – que l'ordre des *appels récursifs* est le même dans le cas du parcours infixe que dans le cas du parcours préfixé. La seule différence tient dans le fait que le traitement est effectué entre les appels, et non pas avant eux. Sur base de cette observation, on adapte aisément l'algorithme itératif donné pour le parcours postfixé, afin d'obtenir un algorithme pour le parcours infixe. L'adaptation consiste à changer le traitement à effectuer quand on rencontre un couple du type $(A, \texttt{d})$ au sommet de la pile : il faut maintenant *pusher* successivement (**FilsDroit**($A$), d), ($A$, t) et (**FilsGauche**($A$), d), afin que la racine soit traitée après le développement de **FilsGauche**($A$), mais avant le développement de **FilsDroit**($A$).

L'Algorithme 34 présente cette solution.

### 7.2.3 Parcours postfixé

Le parcours *postfixé* vient compléter les deux précédents : on y traite la racine *après* avoir traité successivement les deux fils. La version récursive est donnée à l'Algorithme 32.

La version *itérative* est, elle aussi, une modification des parcours préfixé et infixe, dans laquelle on **Push** successivement $(A, \mathtt{t})$, (**FilsDroit**$(A), \mathtt{d})$ et (**FilsGauche**$(A), \mathtt{d})$. Elle est donnée à l'Algorithme 35.

### 7.2.4 Le parcours par niveaux ou parcours en largeur

Le parcours *par niveaux* (parfois appelé *parcours en largeur*) se différencie des trois parcours précédents car il n'a pas de formulation récursive immédiate. Par contre, sa formulation itérative ressemble aux parcours précédents.

Commençons par expliquer intuitivement ce que nous entendons par *parcours par niveaux*. Comme on le voit sur la Fig. 7.7, il s'agit de parcourir d'abord tous les nœuds de profondeur 1, puis ceux de profondeur 2, puis ceux de profondeur 3, etc, et ce de gauche à droite (suivant l'ordre imposé naturellement par les pointeurs).

Si nous considérons l'arbre binaire de la Fig. 7.5, le parcours par niveaux se déroule comme suit. On traite d'abord le nœud $n_1$. Ce faisant, on rencontre ses deux fils $n_2$ et $n_3$, que l'on met en attente. Ensuite, on traite $n_2$. On rencontre alors ses deux fils $n_4$ et $n_5$ qui doivent, eux aussi être mis en attente. Néanmoins, contrairement à ce qui se passait avec la pile utilisée dans les parcours préfixé, infixe ou postfixé, les fils $n_4$ et $n_5$ devront être traité *après* qu'on a traité $n_3$. En effet, il faut avoir traité tous les nœuds de même profondeur avant de passer au « niveau » suivant. Or, $n_3$ est de profondeur 2, alors que $n_4$ et $n_5$ sont de profondeur 3. Ce sont donc les nœuds qui ont été mis en attente en premier qui devront être traités d'abord. On a affaire à une politique de type FIFO, caractéristique des files.

Ceci nous fournit un algorithme pour le parcours en largeur. Celui-ci maintient en permanence une *file* d'éléments à traiter, qui contient initialement l'arbre $A$ à parcourir. Tant que la file n'est pas vide, l'algorithme prélève l'arbre qui se trouve en début de file, traite sa racine, et insère, successivement le sous-arbre gauche et le sous-arbre droit de $A$ dans la file.

```
ParcoursLargeur(Nœud * A)
début
    File F ;
    F :=FileVide() ;
    F := Enqueue(F,A) ;
    tant que ¬EstVide(F) faire
        A := First(F) ;
        F := Dequeue(F) ;
        si ¬EstVide(A) alors
            Traiter A ;
            F := Enqueue(F, FilsGauche(A)) ;
            F := Enqueue(F, FilsDroit(A)) ;
fin
```

**Algorithme 37** : Le parcours par niveaux d'un arbre binaire.

**Exemple 26** *La Fig. 7.8 illustre l'exécution du parcours en largeur sur l'arbre de la Fig. 7.5. À chaque étape, on a représenté la file correspondante.*

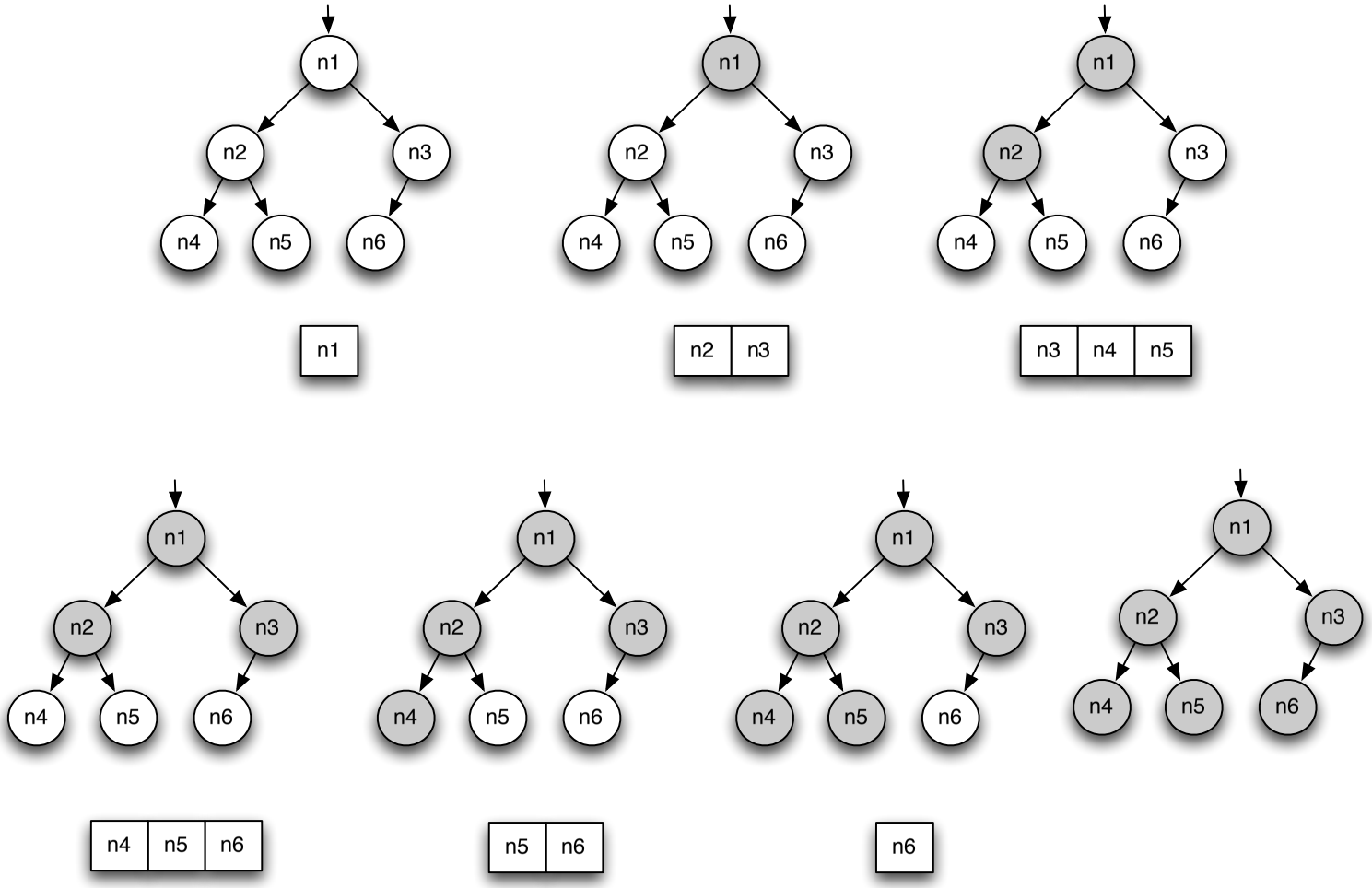



FIG. 7.8 – Une illustration du parcours en largeur. Les nœuds grisés ont été traités. Le contenu de la file à chaque étape est représenté sous l'arbre.

### 7.2.5 Applications des parcours

**Le calcul de la hauteur d'un arbre binaire** Afin d'obtenir un algorithme récursif calculant la hauteur d'un arbre, il faut d'abord en donner une définition récursive. Naturellement, la hauteur de l'arbre vide est 0. Si nous voyons un arbre comme une racine flanquée de deux sous-arbres, il est clair que la hauteur de l'arbre dépendra des hauteurs des sous-arbre, auxquelles il faudra ajouter 1 pour la racine. Comme la hauteur d'un arbre est définie par rapport à son *plus long chemin*, il convient de garder la hauteur du *plus haut* des deux sous-arbre :

**Définition 11** *La hauteur* hauteur$(A)$ *d'un arbre A est :*
- *0 si l'arbre est vide*
- $1 + \max\{\text{hauteur}((\textbf{\textit{FilsGauche}}(A)), \text{hauteur}((\textbf{\textit{FilsDroit}}(A))\}$, *sinon*

On voit qu'il faut donc calculer la hauteur des deux fils avant de pouvoir calculer la hauteur de l'arbre. Il s'agit d'un parcours *postfixé*, comme le montre l'Algorithme 38.

```
Entier Hauteur(Nœud * A)
début
    si EstVide(A) alors
        retourner 0 ;
    sinon
        Entier hg, hd ;
        hg := Hauteur(FilsGauche(A)) ;
        hd := Hauteur(FilsDroit(A)) ;
        retourner 1 + max{hg, hd} ;
fin
```

**Algorithme 38** : Un algorithme récursif pour calculer la hauteur d'un arbre binaire.

**L'évaluation d'une expression** Considérons à nouveau un arbre d'expression, et écrivons un algorithme qui calcule *la valeur de l'expression* représentée par l'arbre. Si nous regardons l'exemple 7.6, nous constatons qu'il ne nous est pas possible d'évaluer immédiatement la valeur de la multiplication représentée dans la racine. En effet, il faut d'abord connaître les valeurs des expressions stockées dans les fils gauche et droit. De même, pour évaluer la valeur de l'addition stockée dans le fils gauche de la racine, il faut d'abord connaître les valeurs des deux feuilles 3 et 5. Par contre, la valeur d'une feuille est directement stockée dans son étiquette.

On obtient ainsi un algorithme récursif pour évaluer un arbre d'expression :

1. Si l'arbre est vide, la valeur est 0 ;
2. Si l'arbre est une feuille $n$, la valeur est $\Lambda(n)$ ;
3. Sinon, la racine $n$ de l'arbre possède deux fils $fg$ et $fd$, et la valeur de l'expression est $fg\ \Lambda(n)\ fd$ (où $\Lambda(n)$ est l'opérateur stocké dans la racine).

Il s'agit à nouveau d'un parcours postfixé, puisqu'il faut évaluer les fils avant de pouvoir évaluer la racine. Cet algorithme est donné en détail à l'Algorithme 39

```
Entier ÉvaluationExpression(Nœud * A)
début
    si EstVide(A) alors
        retourner 0 ;
    sinon si EstFeuille(A) alors
        retourner InfoRacine(A) ;
    sinon
        Entier vg, vd ;
        vg := ÉvaluationExpression(FilsGauche(A)) ;
        vd := ÉvaluationExpression(FilsDroit(A)) ;
        si InfoRacine(A) = + alors
            retourner vg + vd ;
        sinon si InfoRacine(A) = − alors
            retourner vg − vd ;
        sinon si InfoRacine(A) = ∗ alors
            retourner vg ∗ vd ;
        sinon
            retourner vg/vd ;
fin
```

**Algorithme 39** : Un algorithme récursif pour évaluer un arbre d'expression.

**L'affichage d'une expression** Regardons maintenant comment nous pouvons *afficher* (en notation conventionnelle) une expression stockée dans un arbre (en insérant des parenthèses afin de garantir que la priorité exprimée par l'arbre est respectée).

À nouveau, nous allons analyser le problème sous l'angle *récursif*. Il est clair que l'affichage de l'arbre vide, ne produit rien, et que l'affichage d'une feuille revient à afficher son étiquette. Si, par contre, nous avons affaire à un arbre dont la racine possède des fils, il convient d'afficher d'abord l'expression correspondant au fils gauche, puis l'opérateur stocké dans la racine, et enfin l'expression que représente le fils droit. Il faut prendre garde à entourer cet affichage de parenthèses ouvrantes et fermantes, afin de s'assurer que la priorité est respectée. En effet, sans cela, l'algorithme afficherait $3+5*2$ quand il est exécuté sur l'arbre de la Fig. 7.6, ce qui n'est pas correct. Avec les parenthèses, on affiche $((5+3)*2)$.

Nous obtenons donc une variation du parcours infixe, puisque le contenu de la racine est affiché, entre les deux appels récursifs qui traitent les fils gauche et droit. L'Algorithme 40 présente cette solution.

```
Entier AffichageExpression(Nœud * A)
début
    si ¬EstVide(A) alors
        si EstFeuille(A) alors
            Afficher Info(A) ;
        sinon
            Afficher '(' ;
            AffichageExpression(FilsGauche(A)) ;
            Afficher Info(A) ;
            AffichageExpression(FilsGauche(D)) ;
            Afficher ')' ;
fin
```

**Algorithme 40** : Un algorithme récursif pour afficher une expression stockée dans un arbre