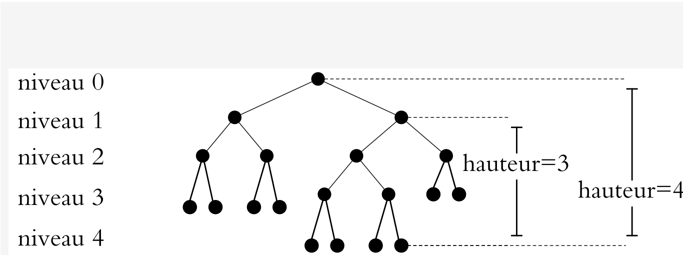
6 Propriétés des arbres



Niveau (*level/depth*) d'un nœud u : longueur du chemin qui mène à u à partir de la racine

Hauteur (*height*) d'un nœud u : longueur maximale d'un chemin dans le sous-arbre de u

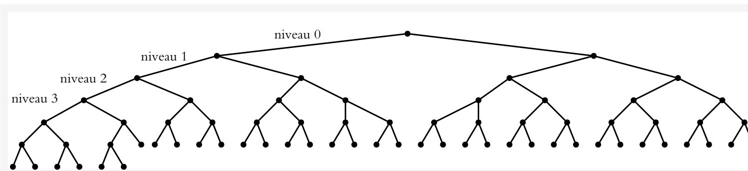
Hauteur de l'arbre : hauteur de la racine (= niveau maximal de nœuds)

Longueur du chemin (interne/externe) (*internal/external path length*) somme des niveaux de tous les nœuds (internes/externes)

Théorème 6.1. *Un arbre binaire à n nœuds externes contient $(n-1)$ nœuds internes.*

Théorème 6.2. *La hauteur h d'un arbre binaire à n nœuds externes est bornée par $\lceil \lg(n) \rceil \le h \le n-1$.*

Démonstration. Un arbre de hauteur $h = 0$ ne contient qu'un seul nœud externe, et les bornes sont correctes. Pour $h > 0$, on définit m_k comme le nombre de nœuds internes au niveau $k = 0, 1, 2, \ldots, h-1$ (il n'y a pas de nœud interne au niveau h). Par Théorème 6.1, on a $n - 1 = \sum_{k=0}^{h-1} m_k$. Comme $m_k \ge 1$ pour tout $k = 0, \ldots, h-1$, on a que $n - 1 \ge \sum_{k=0}^{h-1} 1 = h$. Pour une borne supérieure, on utilise que $m_0 = 1$, et que $m_k \le 2m_{k-1}$ pour tout $k > 0$. En conséquence, $n - 1 \le \sum_{k=0}^{h-1} 2^k = 2^h - 1$, d'où $h \ge \lg n$. La preuve montre aussi les arbres extrêmes : une chaîne de nœuds pour $h = n - 1$, et un arbre binaire complet. ■



Un **arbre binaire complet** de hauteur h : il y a 2^i nœuds à chaque niveau $i = 0, \ldots, h-1$. On «remplit» les niveaux de gauche à droit.

Exercice 6.1. *Soit n_k le nombre de nœuds avec k enfants dans un arbre ordonné ($k = 0, 1, 2, \ldots$). Démontrer que*

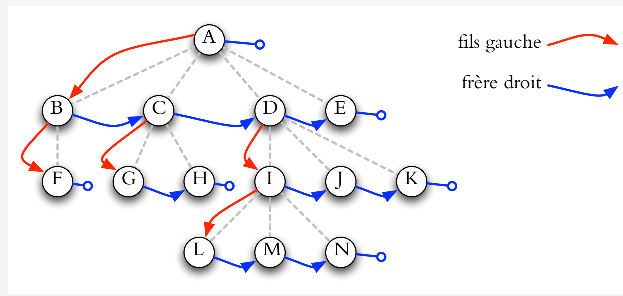
$$n_0 = 1 + n_2 + 2 \cdot n_3 + 3 \cdot n_4 + \cdots + (d-1) \cdot n_d,$$

où d est le degré maximal dans l'arbre. Notez que Théorème 6.1 est un cas spécial de cette égalité avec $d = 2$. ***Indice :*** *compter les nœuds deux fois — une fois selon le nombre d'enfants, et une fois selon le parent.*

6.1 Représentation d'un arbre

```
class TreeNode
{
   TreeNode parent;        // null pour la racine
   TreeNode enfant_gauche; // null si noeud externe
   TreeNode enfant_droit;  // null si noeud externe
   // ... d'autre information
}
```

Arbre = ensemble d'objets représentant de nœuds + relations parent-enfant. En général, on veut retrouver facilement le parent et les enfants de n'importe quel nœud. Souvent, les nœuds externes ne portent pas de données, et on les représente simplement par des liens `null`.
Si l'arbre est d'arité k, on peut avoir un tableau `TreeNode[] enfants` de taille `enfants.length` $= k$.



Si l'arité de l'arbre n'est pas connu en avance (ou la plupart des nœuds ont très peu d'enfants), on peut utiliser une liste pour stocker les enfants : c'est la représentation **premier fils**, **prochain frère** (*first-child, next-sibling*). (Le premier fils est la tête de la liste des enfants, et le prochain frère est le pointeur au prochain nœud sur la liste des enfants.)

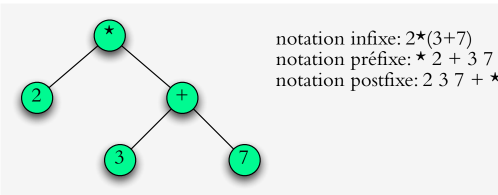
6.2 Récurrences avec un arbre

La structure récursive de l'arbre permet des solutions naturelles par récurrences.

```
    Algo HAUTEUR(x)              // calcule la hauteur du nœud x
H1  max ← −1                     // (hauteur maximale des enfants)
H2  si x est interne alors
H3      pour tout enfant y de x faire
H4          h ← HAUTEUR(y) ;
H5          si h > max alors max ← h
H6  retourner 1 + max            // (visite postfixe)
```

```
    Algo NIVEAU(x, n)
    // parent de x est à niveau n
    // appel initiel avec x = racine et n = −1
N1  si x est interne alors
N2      niveau[x] ← n + 1        // (visite préfixe)
N3      pour tout enfant y de x faire
N4          NIVEAU(y, n + 1)
```

6.3 Arbre syntaxique



notation infixe: 2⋆(3+7)
notation préfixe: ⋆ 2 + 3 7
notation postfixe: 2 3 7 + ⋆

Une expression arithmétique peut être representée par un **arbre syntaxique**. Parcours différents du même arbre mènent à des représentations différentes de la même expression. (L'arbre montre l'application de règles dans une grammaire formelle pour expressions : $E \to E + E | E * E | \mathit{nombre}$).

Une opération arithmétique $a \,\mathrm{op}\, b$ est écrite en **notation polonaise inverse** ou notation «postfixée» par $a\, b\, \mathrm{op}$. Avantage : pas de parenthèses ! Exemples : $1 + 2 \to 1\, 2 +$, $(3 - 7) * 8 \to 3\, 7\ -\ 8\, *$. Une telle expression s'évalue à l'aide d'une pile : $\mathrm{op} \leftarrow \mathsf{pop}()$, $b \leftarrow \mathsf{pop}()$, $a \leftarrow \mathsf{pop}()$, $c \leftarrow \mathrm{op}(a, b)$, $\mathsf{push}(c)$. On répète le code tandis qu'il y a un opérateur en haut. À la fin, la pile ne contient que le résultat numérique. L'évaluation correspond à un parcours postfixe de l'arbre syntaxique. W(fr)

```
    Algorithme EVAL(x)           // (évalation de l'arbre syntaxique avec racine x)
E1  si x n'a pas d'enfants alors retourner sa valeur // (c'est une constante)
E2  sinon // (x est une opération op d'arité k)
E3      pour i ← 0, ..., k − 1 faire f_i ← EVAL(x.enfant[i])
E4      retourner le résultat de l'opération op avec les opérands (f_0, f_1, ..., f_{k−1})
```

Langages. La notation préfixe est généralement utilisée pour les appels de procédures, fonctions ou de méthodes dans des langages de programmation populaires (comme Java et C). En même temps, les opérations arithmétiques et logiques sont typiquement écrites en notation infixe.

PostScript est un langage de programmation qui utilise la notation postfixe à l'aide d'une pile. Toutes les fonctions et commandes de contrôle enlèvent leurs arguments de la pile et y placent les valeurs retournées (s'il y en a). P.e., le code "`b {5 2 sub 20 moveto 30 40 lineto} if`" dessine une ligne entre les points $(3, 20)$ et $(30, 40)$ si b est vrai. W(fr)