6 Arbres

6.1 Structures arborescentes

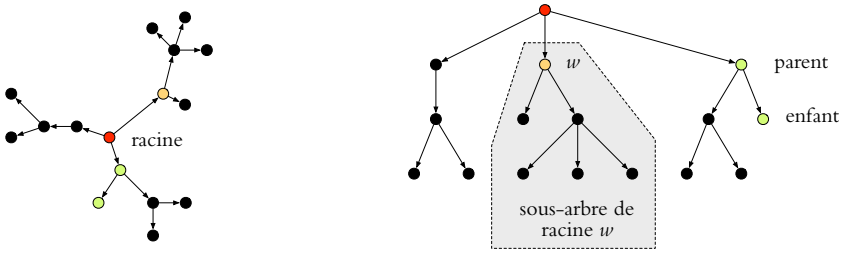
Un arbre est une structure récursive d'importance fondamentale dans beaucoup de domaines d'informatique :

- ⋆ les structures arborescentes s'adaptent à l'abstraction de nombreux problèmes réels sur hiérarchies et réseaux ;
- ⋆ les meilleures structures de données sont souvent les réalisations concrètes d'arbres ;
- ⋆ on utilise des arbres pour décrire les propriétés des algorithmes récursifs dans l'analyse théorique ;
- ⋆ on compile le code source à l'aide des arbres de syntaxe ;
- ⋆ ...

Dans ce cours, on considère les arbres enracinés (ou l'ordre des enfants n'est pas important) et les arbres ordonnés (comme l'arbre binaire, ou les enfants sont ordonnés dans une liste). On identifie un abre par sa racine.

Définition 6.1. *Un **arbre enraciné** T ou arboresence est une structure définie sur un ensemble de nœuds qui*

1. *est un **nœud externe**, ou*
2. *est composé d'un **nœud interne** appellé la **racine** r, et un ensemble d'arbres enracinés (les **enfants**)*



Définition 6.2. *Un **arbre ordonné** T est une structure définie sur un ensemble de nœuds qui*

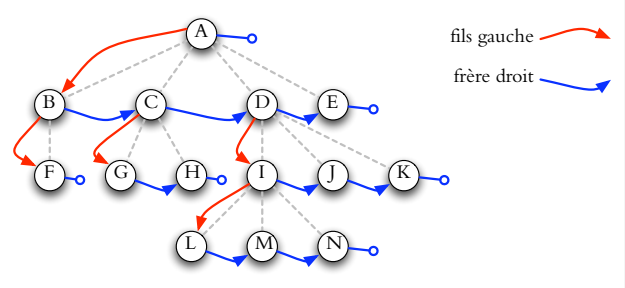
1. *est un **nœud externe**, ou*
2. *est composé d'un **nœud interne** appellé la **racine** r, et les arbres $T_0, T_1, T_2, \ldots, T_{d-1}$. La racine de T_i est apellé l'**enfant** de r étiqueté par i ou le i-ème enfant de r.*

*Le **degré** (ou **arité**) d'un nœud est le nombre de ses enfants : les nœuds externes sont de degré 0. Le degré de l'arbre est le degré maximal de ses nœuds. Un arbre k-aire est un arbre ordonné où chaque nœud interne possède exactement k enfants. Un **arbre binaire** est un arbre ordonné où chaque nœud interne possède exactement 2 enfants : les sous-arbres gauche et droit.* W(fr)

6.2 Représentation d'un arbre

```
class TreeNode // noeud interne dans arbre binaire
{
   TreeNode parent; // null pour la racine
   TreeNode left;   // enfant gauche, null=enfant externe
   TreeNode droit;  // enfant droit, null=enfant externe
   // ... d'autre information
}
class MultiNode // noeud interne dans arbre ordonné
{
   TreeNode parent; // null pour la racine
   TreeNode[] children; // enfants; children.length=arité
   // ... d'autre info
}
```

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 les stocker dans un tableau de taille k.

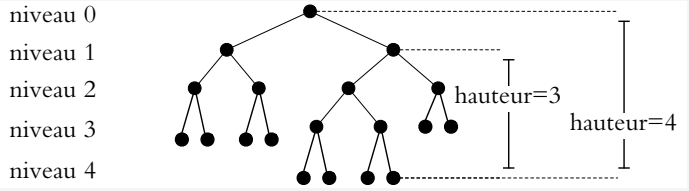


Si l'arité de l'arbre n'est pas connu en avance (ou la plupart des nœuds ont très peu d'enfants), on peut utiliser une liste pour stocker les enfants : c'est la représentation **premier fils**, **prochain frère** (*first-child, next-sibling*). (Le premier fils est la tête de la liste des enfants, et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

Théorème 6.1. *Il existe une correspondance 1-à-1 entre les arbres binaires à n nœuds internes et les arbres ordonnés à n nœuds (internes et externes).*

Démonstration. On peut intérprêter «premier fils» comme «enfant gauche», et «prochain frère» comme «enfant droit» pour obtenir un arbre binaire unique qui correspond à un arbre ordonné arbitraire, et vice versa. ■

6.3 Propriétés



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

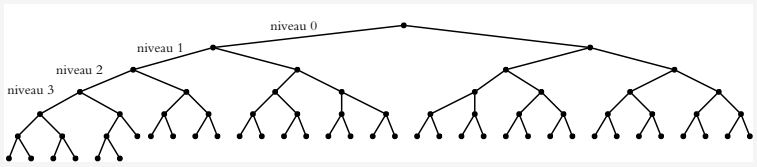
Hauteur (*height*) d'un nœud u : longueur maximale d'un chemin de u jusqu'à un nœud externe dans le sous-arbre de u

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)

$$\mathsf{hauteur}[x] = \begin{cases} 0 & \text{si } x \text{ est externe ;} \\ 1 + \max_{y \in x.\mathsf{children}} \mathsf{hauteur}[y] \end{cases}$$

$$\mathsf{niveau}[x] = \begin{cases} 0 & \text{si } x \text{ est la racine } (x.\mathsf{parent} = \mathsf{null}) ; \\ 1 + \mathsf{niveau}[x.\mathsf{parent}] & \text{sinon} \end{cases}$$



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.

Théorème 6.2. *Un arbre binaire à n nœuds externes contient $(n-1)$ nœuds internes.*

Théorème 6.3. *La hauteur h d'un arbre binaire à n nœuds externes est bornée par $\lceil \lg(n) \rceil \leq h \leq n-1$.*

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 6.2, on a $n - 1 = \sum_{k=0}^{h-1} m_k$. Comme $m_k \geq 1$ pour tout $k = 0, \ldots, h-1$, on a que $n - 1 \geq \sum_{k=0}^{h-1} 1 = h$. Pour une borne supérieure, on utilise que $m_0 = 1$, et que $m_k \leq 2m_{k-1}$ pour tout $k > 0$. En conséquence, $n - 1 \leq \sum_{k=0}^{h-1} 2^k = 2^h - 1$, d'où $h \geq \lg n$. La preuve montre aussi les arbres extrêmes : une chaîne de nœuds pour $h = n - 1$, et un arbre binaire complet. ■

6.4 Parcours

Un parcours visite tous les nœuds de l'arbre. Dans un **parcours préfixe** (*preorder traversal*), chaque nœud est visité avant que ses enfants soient visités. On calcule ainsi des propriétés avec récurrence vers le parent (comme niveau). Dans un **parcours postfixe** (*postorder traversal*), chaque nœud est visité après que ses enfants sont visités. On calcule ainsi des propriétés avec récurrence vers les enfants (comme hauteur).

Dans les algorithmes suivants, un nœud externe est null, et chaque nœud interne N possède les variables N.children (si arbre ordonné), ou N.left et N.right (si arbre binaire). L'arbre est stocké par une référence à sa racine root.

```
Algo PARCOURS-PRÉFIXE(x)
1  if x ≠ null then
2      «visiter» x
3      for y ∈ x.children do
4          PARCOURS-PRÉFIXE(y)
```

```
Algo PARCOURS-POSTFIXE(x)
1  if x ≠ null then
2      for y ∈ x.children do
3          PARCOURS-POSTFIXE(y)
4      «visiter» x
```

```
Algo NIVEAU(x, n)          // remplit niveau[···]
// parent de x est à niveau n
// appel initiel avec x = root et n = −1
N1  if x ≠ null then
N2      niveau[x] ← n + 1          // (visite préfixe)
N3      for y ∈ x.children do
N4          NIVEAU(y, n + 1)
```

```
Algo HAUTEUR(x)                    // retourne hauteur de x
H1  max ← −1                       // (hauteur maximale des enfants)
H2  if x ≠ null then
H3      for y ∈ x.children do
H4          h ← HAUTEUR(y) ;
H5          if h > max then max ← h
H6  return 1 + max                 // (visite postfixe)
```

Lors d'un **parcours infixe** (*inorder traversal*), on visite chaque nœud après son enfant gauche mais avant son enfant droit. (Ce parcours ne se fait que sur un arbre binaire.)

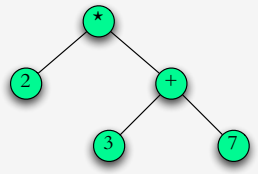
```
Algo PARCOURS-INFIXE(x)
1  if x ≠ null then
2      PARCOURS-INFIXE(x.left)
3      «visiter» x
4      PARCOURS-INFIXE(x.right)
```

Un parcours préfixe ou postfixe peut se faire aussi à l'aide d'une pile. Si au lieu de la pile, on utilise une queue, alors on obtient un **parcours par niveau**.

Algo PARCOURS-PILE
1 initialiser la pile P
2 P.push(root)
3 **while** $P \neq \emptyset$
4 $\quad x \leftarrow P$.pop()
5 $\quad$ **if** $x \neq$ null **then**
6 $\quad\quad$ «visiter» x
7 $\quad\quad$ **for** $y \in x$.children **do** P.push(y)

Algo PARCOURS-NIVEAU
1 initialiser la queue Q
2 Q.enqueue(root)
3 **while** $Q \neq \emptyset$
4 $\quad x \leftarrow Q$.dequeue()
5 $\quad$ **if** $x \neq$ null **then**
6 $\quad\quad$ «visiter» x
7 $\quad\quad$ **for** $y \in x$.children **do** Q.enqueue(y)

6.5 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 : $E \rightarrow E + E | E * E |$ *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 : $1 + 2 \rightarrow 1\,2\,+$, $(3 - 7) * 8 \rightarrow 3\,7\,-\,8\,*$. Une telle expression s'évalue à l'aide d'une pile : op $\leftarrow$ pop(), $b \leftarrow$ pop(), $a \leftarrow$ pop(), $c \leftarrow \mathrm{op}(a, b)$, 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 $\quad$ **pour** $i \leftarrow 0, \ldots, k-1$ **faire** $f_i \leftarrow$ EVAL(x.enfant[i])
E4 $\quad$ **retourner** le résultat de l'opération op avec les opérands $(f_0, f_1, \ldots, f_{k-1})$

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. Le langage **PostScript** (Devoir 2) utilise la notation postfixe partout, même en instructions de contrôle. En conséquence, l'interpréteur se relie sur des piles dans l'exécution. Le code W(fr)

```
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. En Javaesque, on écrirait

```
if (b) {moveTo(5-2,20); lineTo(30, 40)}.
```

Le langage **Lisp** utilise la notation préfixe avec parènthèses obligatoires. En conséquence, les listes sont des objets fondamentaux ("Lisp"=*List Processing Language*) lors d'exécution : une liste est formée d'une tête (car pour Lispéens) et d'une queue — cette dernière est aussi une liste (cdr pour le Lispéen). W(fr)

```
(with-canvas (:width 100 :height 200) ((moveto (- 5 2) 20) (lineto 30 40)))
```