5 Récursion et arbres

5.1 Diviser pour régner

La technique de «**diviser pour régner**» (*divide-and-conquer*) exploîte la nature récursive de solutions optimales à une classe de problèmes. La démarche générale est de (1) couper le problème dans des sous-problèmes similaires, (2) chercher la solution optimale aux sous-problèmes (récursion), (3) combiner les résultats. Quand un problème de taille n est coupé en m sous-problèmes de taille n_i: $i = 1, \ldots, m$, le temps de calcul est déterminé par la récurrence $T(n) = \tau_{\text{couper}}(n) + \sum_{i=1,\ldots,m} T(n_i) + \tau_{\text{combiner}}(n)$, où $\tau_{\text{couper}}(n)$ et $\tau_{\text{combiner}}(n)$ sont les temps pour couper et combiner.

Exemple 5.1. On prend l'example de chercher le maximum dans un tableau.

```
    MAX-DR(x[0..n−1], g, d) // trouve le max parmi x[g..d−1]
    // appel initiel avec g = 0, d = n
D1  if d − g = 0 then return −∞                              // cas de base 0
D2  if d − g = 1 then return x[g]                            // cas de base 1
D3  mid ← ⌊(g + d)/2⌋                                        // diviser
D4  m1 ← MAX-DR(x, g, mid) ; m2 ← MAX-DR(x, mid, d)          // sous-problèmes par récurrence
D5  return max{m1, m2}                                       // combiner
```

Le temps de calcul de l'algorithme MAX-DR s'écrit par la récurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(1) \qquad \{n \geq 2\} \tag{5.1}$$

Théorème 5.1. *La solution de l'Équation* (5.1) *est* $T(n) = \Theta(n)$. ♠

Exemple 5.2. On a fourni une solution au Tours de Hanoï par le principe de «diviser pour régner» (Notes de cours, §1.6). Le temps d'exécution du réarrangement de $n > 0$ disques s'écrit par la récurrence $T(n) = 2T(n-1) + \Theta(1)$ avec solution $T(n) = \Theta(2^n)$. ♠

5.2 Récursion terminale

On peut toujours transformer une boucle en un algorithme récursif équivalent.

```
    MAX-ITER(x[0..n−1])
M1  initialiser max ← −∞
M2  for i ← 0, ..., n−1 do
M3      if x[i] > max then max ← x[i]

M4  return max
```

```
    MAX-TERM(x[0..n−1], i, max)          // variables locales de l'itération
    // appel initiel avec i = 0, max = −∞
T1  if i < n then                        // condition d'arrêt de l'itération
T2      if x[i] > max then max ← x[i]    // corps d'une itération
T3      return MAX-TERM(x, i+1, max)     // boucler
T4  if i = n then return max             // après la boucle
```

La profondeur maximale de la pile d'exécution (§1.3, Notes 01) caractérise un aspect important de l'efficacité d'un algorithme récursif. La récurrence de MAX-DR (Exemple 5.1) nécessite une pile d'exécution de profondeur $\lceil \lg n \rceil$. Ce qui est pire, la récurrence naïve de MAX-REC de l'Exemple 4.3 peut mèner au débordement de la pile d'exécution, à cause de n appels imbriqués. Par contre, si l'appel récursif est en **position terminale**, on n'a pas besoin d'attendre le retour de l'appel. On peut simplement transférer le contrôle sans sauvegarder le contexte : il suffit de remplacer le bloc d'activation (au lieu d'en empiler un nouveau). En conséquence, la profondeur maximale de la pile reste $\Theta(1)$. Les compilateurs modernes détectent des appels terminaux. Après compilation, MAX-ITER et MAX-TERM sont identiques.

W(fr)

5.3 Liste chaînée comme structure recursive

La liste (simplement) chaînée est une structure récursive. Une liste chaînée est construite par l'application des règles suivantes.

$$\langle liste \rangle \rightarrow \mathsf{null} \qquad \text{(liste vide)}$$
$$\langle liste \rangle \rightarrow \big(\langle noeud \rangle, \langle liste \rangle\big) \qquad \text{(liste commence par } \langle noeud \rangle\text{)}$$

Dans d'autres mots, une liste chaînée est soit une référence null, soit une paire d'un nœud (sa tête) et d'une liste.

On peut exploiter la structure récursive lors du parcours.

```
   LENGTH(N)      // calcule le nombre de nœuds à partir de N
N1 if N = null then return 0
N2 else return 1 + LENGTH(N.next).
```

5.4 Arbres

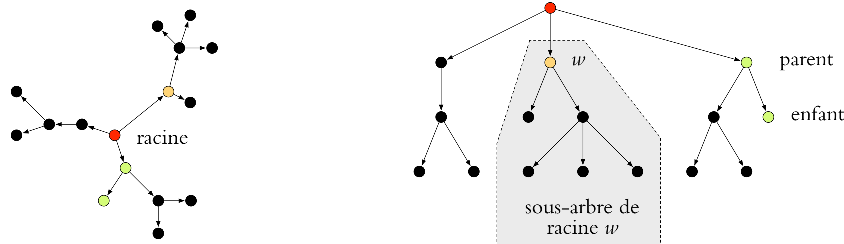
Un arbre est une structure récursive qui joue un rôle central dans la conception et analyse d'algorithmes :

- ⋆ structures de données explicites qui sont des réalisations concrètes d'arbres
- ⋆ arbres pour décrire les propriétés dynamiques des algorithmes récursifs
- ⋆ arbres de syntaxe

On considère des 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).

Définition 5.1. *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 5.2. *Un **arbre ordonné** T est une structure définie sur un ensemble de nœuds qui*

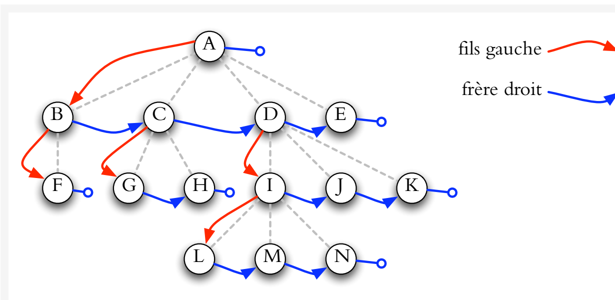
1. *est un **nœud externe**, ou*
2. *est composé d'un **nœud interne** appellé la **racine** r, et les arbres $T_0, T_1, T_2, \ldots, T_{d-1}$. La racine de T_i est apellé l'**enfant** de r étiqueté par i.*

*Le **degré** 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)

5.5 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` $= k$.

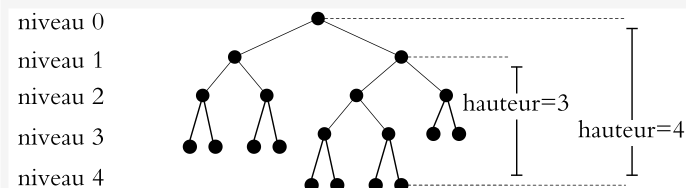


Si l'arité de l'arbre n'est pas connu en avance (ou la plupart des nœuds ont très peu d'enfants), on peut utiliser une liste pour stocker les enfants : c'est la représentation **premier fils**, **prochain frère** (*first-child, next-sibling*). (Le premier fils est la tête de la liste des enfants, et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

Théorème 5.2. *Il existe une correspondance 1-à-1 entre les arbres ordonnés et les arbres binaires.*

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. ∎

5.6 Propriétés et parcours



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)

Définitions récursives :

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

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

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 numeroté), 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 ≠ ∅
4     x ← P.pop()
5     if x ≠ null then
6         «visiter» x
7         for y ∈ x.children do P.push(y)
```

```
Algo PARCOURS-NIVEAU
1 initialiser la queue Q
2 Q.enqueue(root)
3 while Q ≠ ∅
4     x ← Q.dequeue()
5     if x ≠ null then
6         «visiter» x
7         for y ∈ x.children do Q.enqueue(y)
```