Introduction aux arbres

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Les arbres comme objets mathématiques

Définitions

Définition

Un arbre A est un ensemble non vide muni d'une **relation binaire** $\mathcal{R}$ vérifiant :

- $\exists! r \in A,\ \forall x \in A\ \neg(r\ \mathcal{R}\ x)$.
  L'élément r s'appelle la **racine**.
- $\forall x \in A \setminus \{r\}\ \exists! y \in A,\ x\ \mathcal{R}\ y$.
  On dit que y est le **parent** (ou le **père**) de x, et que x est un **fils** de y.
- $\forall x \in A \setminus \{r\}\ \exists n > 0,\ \exists (x_1, \ldots, x_n) \in A^n,$
  $x\ \mathcal{R}\ x_1\ \mathcal{R}\ x_2\ \mathcal{R} \cdots \mathcal{R}\ x_n = r$.

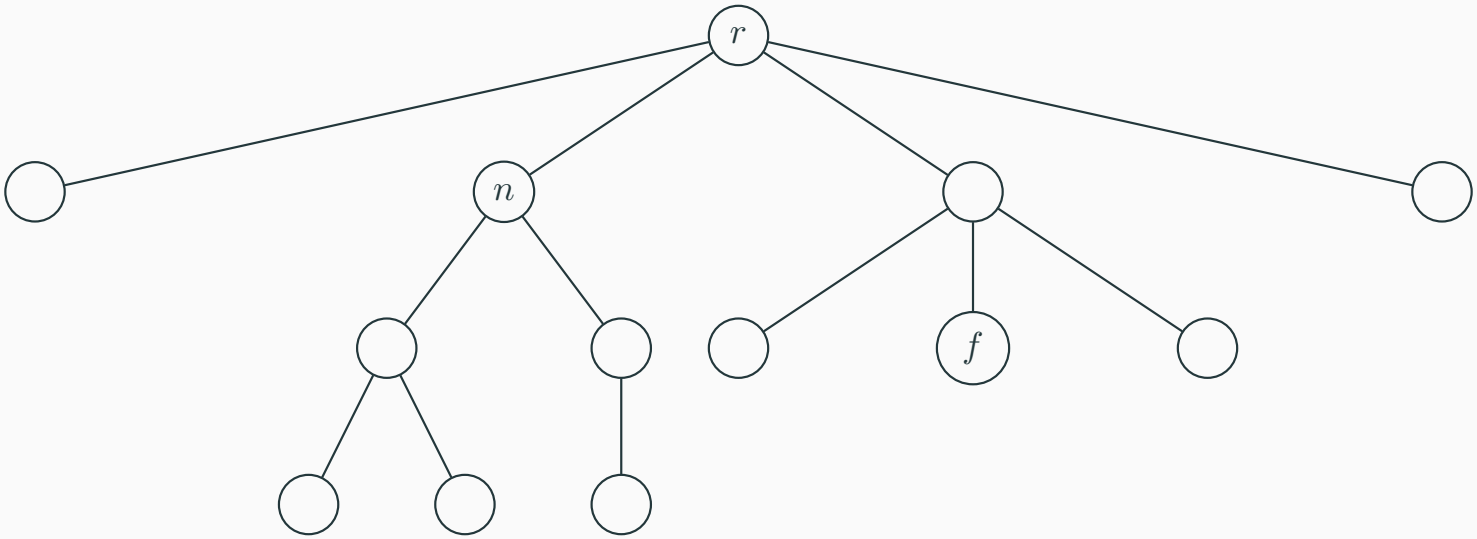
Définitions

Remarque

La relation binaire $x \mathcal{R} y$ signifie que c est un enfant de y.

Les conditions peuvent se résumer ainsi : mis à part la racine r, chaque élément a un unique parent, et en suivant ces liens de parenté on aboutit à la racine.

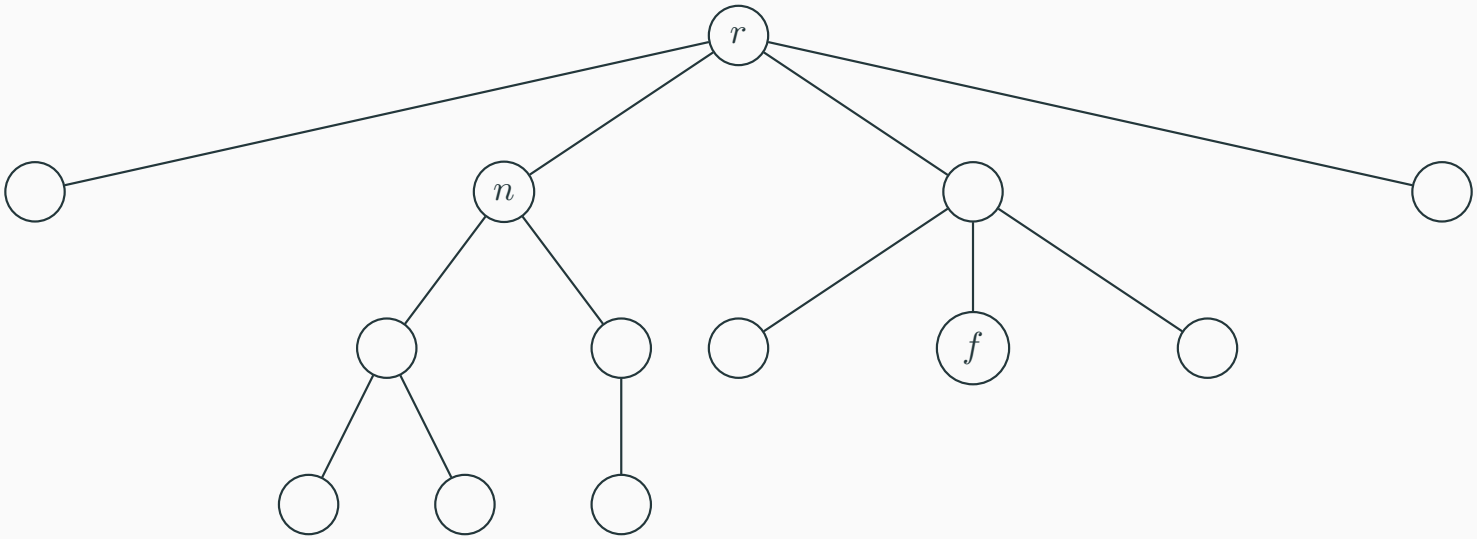
Définitions



Représentation

Visuellement, on peut représenter un arbre avec des nœuds reliés par des arêtes, la racine étant située tout en haut.

Définitions



Définition (feuilles et nœuds internes)

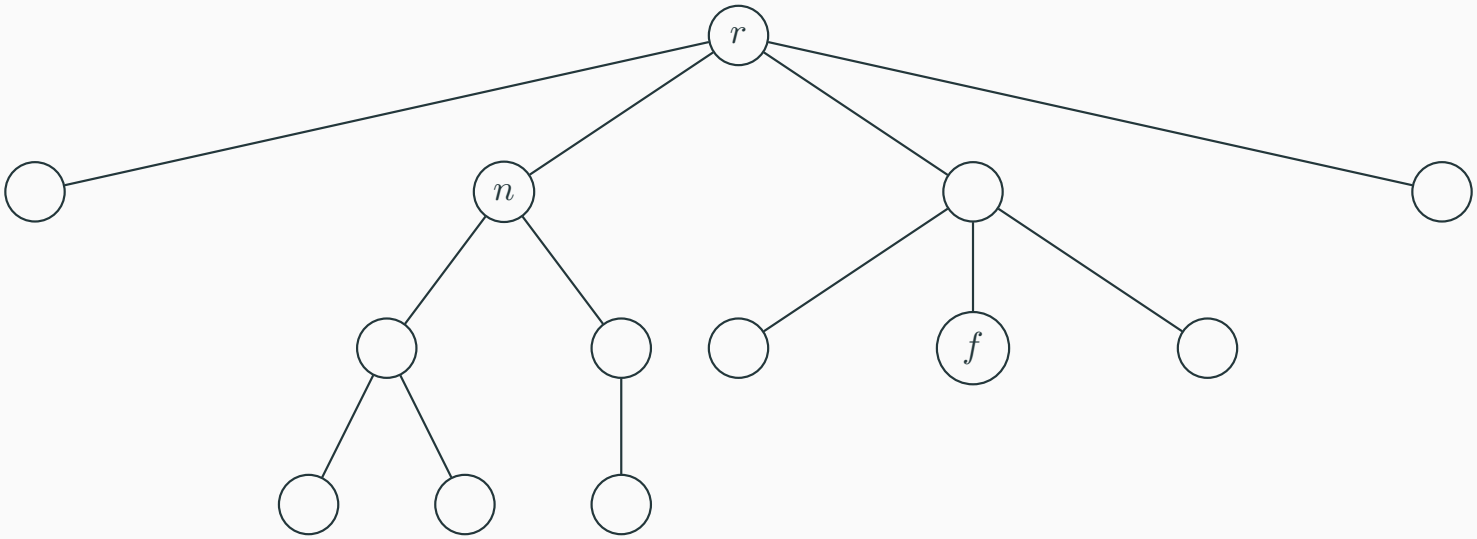
Les éléments de A sont appelés les **nœuds** de l'arbre.

Pour x un nœud, on appelle **arité** de x le nombre de fils de x.

Un nœud d'arité 0 est appelé une **feuille**.

Sinon, c'est un **nœud interne**.

Définitions



Exemple

Dans l'abre ci-dessus, n est un **nœud interne** (d'arité 2), et f est une **feuille**.

Définitions

Définition (arbre binaire)

On appelle **arbre binaire** un arbre dont les nœuds sont d'arité au plus 2, et **arbre binaire entier** un arbre binaire dont les nœuds sont tous d'arité 0 ou 2.

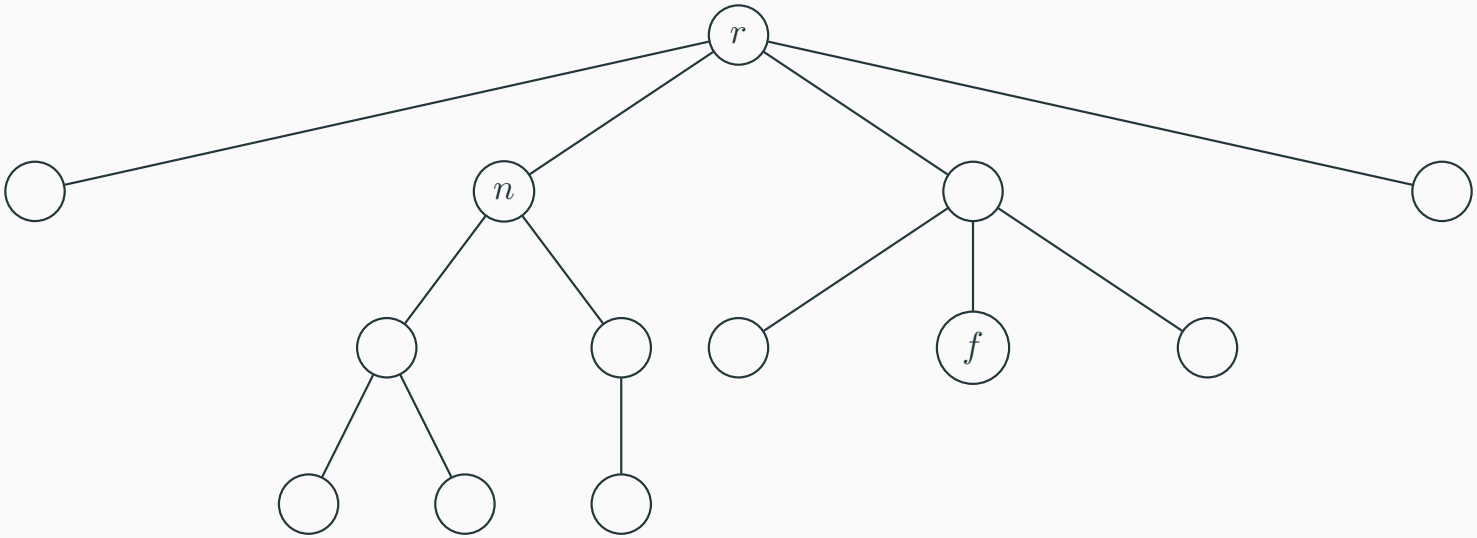
Définitions

Définition (profondeur et hauteur)

Avec r la racine d'un arbre A et x un de ses nœuds, on a vu qu'il existait un unique entier $n \geq 0$ et d'uniques nœuds $x_1, \ldots, x_{n-1}$ tels que $x \mathcal{R} x_1 \mathcal{R} \cdots \mathcal{R} x_{n-1} \mathcal{R} x_n = r$.

- On appelle **profondeur** de x l'entier $n \geq 0$.
- On appelle **hauteur** d'un arbre la profondeur maximale de ses nœuds.

Définitions



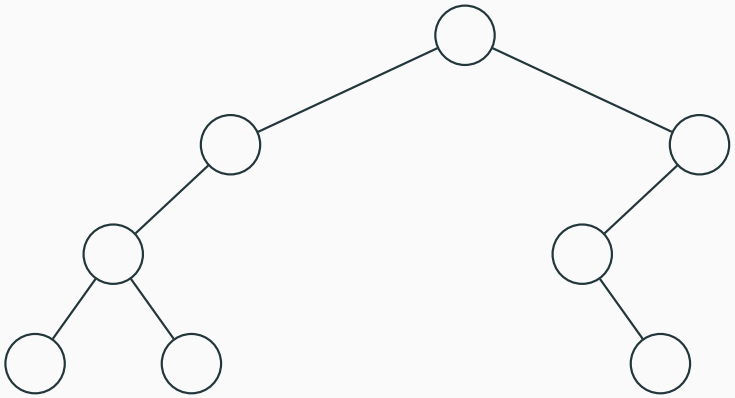
Exemple

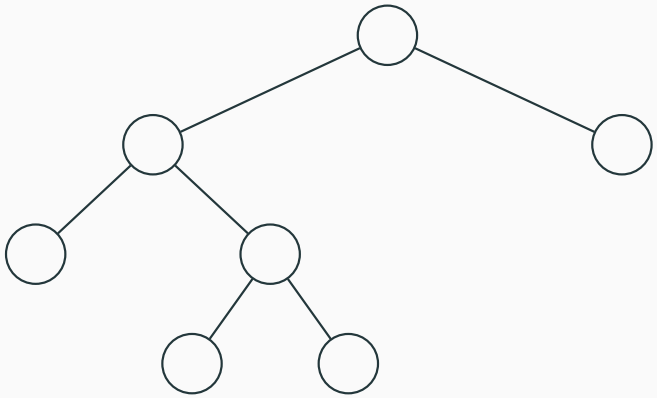
La racine est l'unique nœud à profondeur 0.

Dans l'arbre ci-dessus, n est de profondeur 1, et f est de profondeur 2.

L'arbre est de hauteur 3.

Définitions





Exemple

Les deux arbres ci-dessus sont de hauteur 3.

Définitions

Définition (sous-arbre enraciné)

Soit x un nœud de A. On considère l'ensemble :

$$A_x = \{y \in A \mid \exists n \in \mathbb{N},\ \exists x_1, \ldots, x_{n-1} \in A^{n-1},\ y = x_0\ \mathcal{R}\ x_1\ \mathcal{R} \cdots \mathcal{R}\ x_{n-1}\ \mathcal{R}\ x_n = x\}$$

Alors, on vérifie aisément que la restriction de $\mathcal{R}$ à A_x munit A_x d'une structure d'arbre, de racine x.

Cet arbre se nomme le sous-arbre de A enraciné en x.

On dit aussi que les éléments de A_x forment la descendance de x dans A.

Inégalités entre hauteur et nombre de nœuds

Dénombrement

On donne ici des encadrements faisant intervenir la hauteur et le nombre de nœuds d'un arbre, en fonction de l'arité maximale des nœuds.

Inégalités entre hauteur et nombre de nœuds

Proposition

Pour A un arbre de hauteur h dont les nœuds sont d'arité au plus a, le nombre n de nœuds de A vérifie, si $a > 1$:

$$h + 1 \leq n \leq \frac{a^{h+1} - 1}{a - 1}$$

Preuve

Considérons un nœud à profondeur maximale h. Sur le chemin de ce nœud à la racine, il y a $h + 1$ nœuds, d'où $n \geq h + 1$.

De plus, on montre aisément par récurrence qu'il y a au plus a^p nœuds à profondeur p. On obtient donc l'autre inégalité par somme : $\sum_{p=0}^{h} a^p = \frac{a^{h+1}-1}{a-1}$.

Inégalités entre hauteur et nombre de nœuds

Remarque

Si est un arbre est d'arité maximale 1, il a exactement $h + 1$ nœuds, avec h sa hauteur.

On parle dans ce cas d'arbre **filiforme**.

Inégalités entre hauteur et nombre de nœuds

Corollaire

La hauteur h d'un arbre à n nœuds tous d'arité au plus $a > 1$ vérifie :

$$\log_a ((a-1)n+1) - 1 \leq h \leq n-1$$

Corollaire (cas $a = 2$)

La hauteur h d'un arbre **binaire** à n nœuds vérifie :

$$\lfloor \log_2(n) \rfloor \leq h \leq n-1$$

Preuve

$h+1 \geq \log_2(n+1) > \log_2(n) \geq \lfloor \log_2(n) \rfloor$.

Donc $h+1 > \lfloor \log_2(n) \rfloor$, et ces deux quantités sont des entiers.

Donc $h \geq \lfloor \log_2(n) \rfloor$.

Feuilles et nœuds dans un arbre binaire

Proposition

Un arbre binaire entier ayant p nœuds internes possède $p + 1$ feuilles.

Feuilles et nœuds dans un arbre binaire

Preuve

La démonstration se fait par récurrence forte sur p.

- Si $p = 0$, l'arbre a une seule feuille (sa racine), donc la relation est vérifiée.
- Sinon, soit $p > 0$ et supposons la propriété vraie pour tout $p' < p$. Considérons un arbre A ayant p nœuds internes. La racine étant un nœud interne, notons alors n_g et n_d le nombre de nœuds internes des sous-arbres gauche et droit de la racine.

Feuilles et nœuds dans un arbre binaire

Preuve

Ces sous-arbres sont également binaires entiers, et vérifient $n_g < p$ et $n_d < p$, donc par hypothèse de récurrence, ils ont respectivement $n_g + 1$ et $n_d + 1$ feuilles. Dans A, il y a donc $n_g + n_d + 1$ nœuds internes, et $(n_g + n_d + 1) + 1$ feuilles, donc la propriété est vraie pour p.

Ainsi, par principe de récurrence, la propriété est démontrée.

Feuilles et nœuds dans un arbre binaire

Corollaire

Dans un arbre binaire à p nœuds internes, il y a au plus $p + 1$ feuilles.

Preuve

Rajouter un fils (une feuille) aux nœuds d'arité 1 transforme l'arbre en arbre binaire entier, sans changer le nombre de nœuds internes.

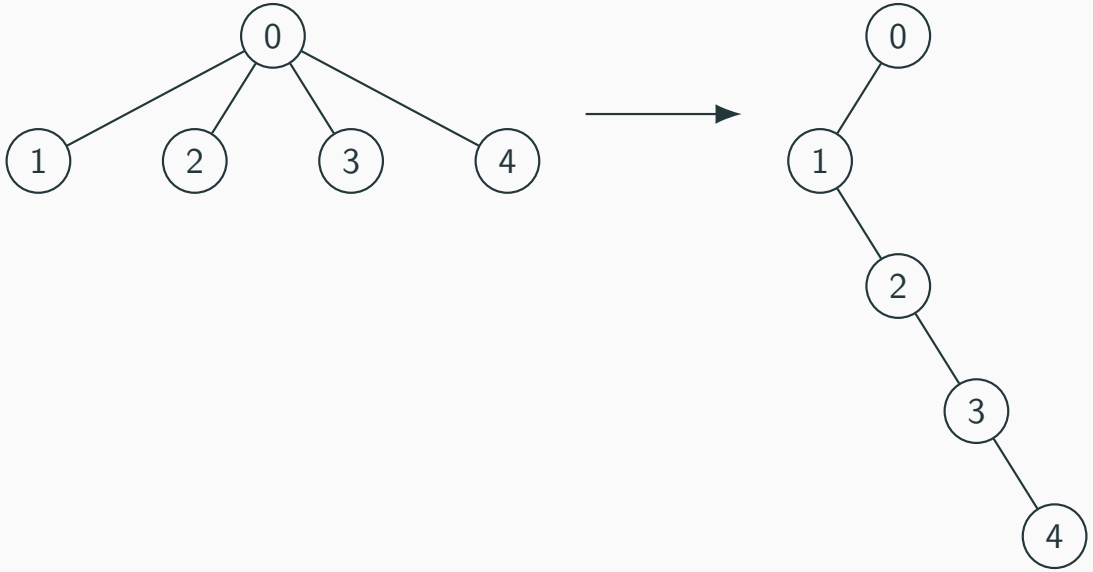
Transformer un arbre d'arité quelconque en un arbre binaire

Transformer un arbre d'arité quelconque en un arbre binaire

Transformation

On peut toujours transformer un arbre d'arité quelconque en un arbre binaire, avec la transformation suivante :

- on ne change pas la racine ;
- pour tout nœud ayant un **frère droit**, son **premier frère droit** devient son **fils droit**.



Les arbres en OCaml

Les arbres en informatique

En informatique

En informatique, les arbres sont utilisés pour stocker de l'information : à chaque nœud est attaché une **étiquette**, qui peut être un entier, une chaîne de caractères, voire même un couple.

De plus, les fils d'un nœud sont en général ordonnés : par exemple pour un arbre binaire entier, on parlera du fils gauche et du fils droit d'un nœud interne.

Les arbres en informatique

Exemple : Système de fichiers

Les fichiers de votre ordinateur sont organisés dans une structure arborescente.

Sous Windows, la racine du système s'appelle `C:\`.
Sous Linux ou Mac OS, elle s'appelle `/`.

Les dossiers sont les nœuds internes, et les fichiers sont les feuilles.

Les arbres en informatique

Exemple : Système de fichiers

Le chemin absolu d'un nœud est son chemin d'accès depuis la racine, où les noms des nœuds sont séparés par des `\` (sous Windows) ou des `/` (sous Linux ou Mac OS).

Le chemin relatif suit le même principe pour expliquer comment aller du dossier courant vers un autre nœud, où on peut utiliser `..` dans l'écriture du chemin pour indiquer qu'il faut remonter au parent.

Les arbres en informatique

Exemple : Arbre syntaxique

Lorsque vous écrivez une expression dans un langage de programmation, elle est représentée en interne par son **arbre syntaxique** (qui permet de représenter comment est parenthésée l'expression).

Les arbres en informatique

Exemple : XML

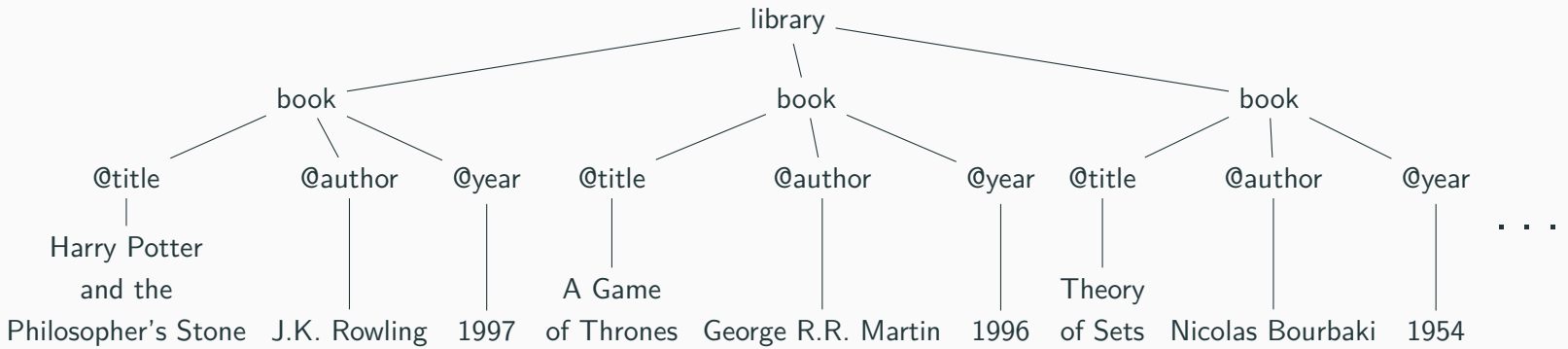
Le langage XML (eXtended Markup Language), créé à la fin des années 90, permet de représenter des données de manière structurée.

Les documents XML sont le format standard pour partager des données via internet, et sont des fichiers textes dont la structure sous-jacente est une structure d'arbre.

Le langage HTML est un exemple de format XML.

XML

```
<library>
  <book title="Harry Potter and the Pilosopher's stone"
    author="J.K. Rowling" year="1997"/>
  <book  title="A Game of Thrones"
    author="George R.R. Martin" year="1996"/>
  <book title="Theory of Sets"
    author="Nicolas Bourbaki" year="1954"/>
  ...
</library>
```



Les arbres en OCaml

OCaml

On propose dans cette section une implémentation **persistante** des arbres : comme pour les listes chaînées, les fonctions sur la structure d'arbre renverront de nouveaux arbres plutôt que de les modifier.

Les arbres en OCaml

```
1  type 'a arbre = N of 'a * 'a arbre list ;;
```

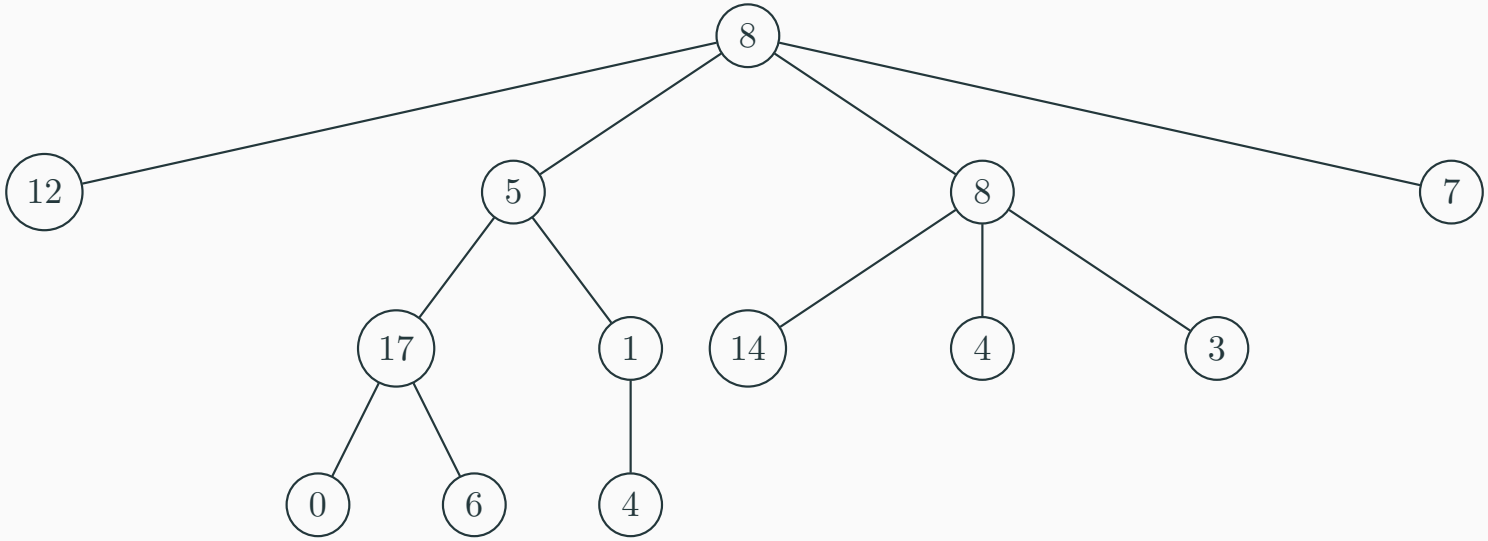
Arbres généraux

Pour les arbres généraux, on peut représenter un nœud par la liste de ses fils, qui sont eux même des arbres.

On distinguera alors une feuille et un nœud interne suivant si la liste des fils est vide ou non.

Pour que les nœuds de l'arbre puissent porter des étiquettes d'un certain type, on peut définir le type (récursif) ci-dessus.

Les arbres en OCaml



Exemple d'arbre à étiquettes entières

```
# ex_arbre ;;
- : int arbre =
N (8,
  [N (12, []);
   N (5, [N (17, [N (0, []); N (6, [])]); N (1, [N (4, [])])]);
   N (8, [N (14, []); N (4, []); N (3, [])]);
   N (7, [])])
```

Les arbres en OCaml

Parcours

Pour parcourir un tel arbre, on utilise en général deux fonctions : l'une qui prend en entrée un arbre, et l'autre une liste d'arbres.

Elles vont s'appeler l'une l'autre, et donc être **mutuellement récursives**.

Les arbres en OCaml

```
1  let rec hauteur a = match a with
2    | N(_,l) -> 1 + max_h l
3  and max_h l = match l with
4    | [] -> -1
5    | x::q -> max (hauteur x) (max_h q)
6  ;;
```

Exemple

Voici un exemple de fonction qui parcourt un arbre, pour calculer sa hauteur.

Les arbres en OCaml

```
1  type ('a, 'b) arbre = F of 'a
2                      | N of 'b * ('a, 'b) arbre list ;;
```

Différents types d'étiquettes

Si on veut faire la distinction entre feuilles et nœuds internes (et leur donner des étiquettes de type différents), on peut utiliser par exemple le type ci-dessus.

Arbres binaires entiers

Les arbres qu'on va manipuler seront souvent des arbres **binaires** (entiers ou non).

On va donc utiliser une implémentation un peu moins générale que celle de la section précédente.

Pour les arbres binaires entiers, un arbre (informatique) est :

- soit une feuille ;
- soit la donnée d'une étiquette, et de deux arbres (ses sous-arbres gauche et droit).

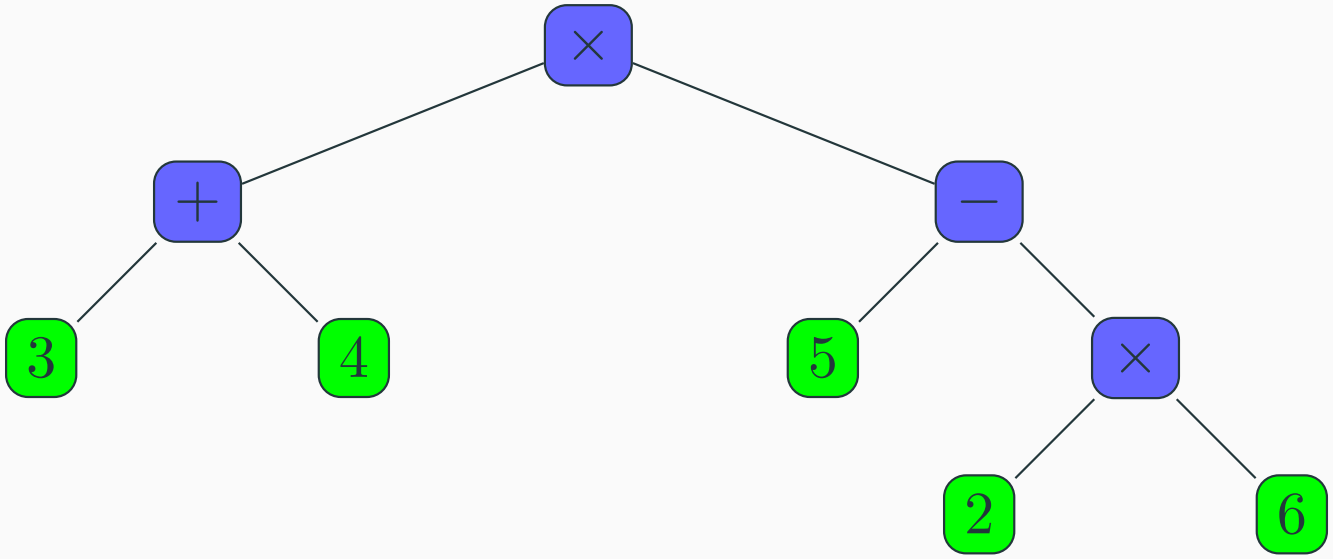
Arbres binaires entiers

```
1  type ('a, 'b) arbre = F of 'a
2                      | N of 'b * ('a, 'b) arbre * ('a, 'b) arbre ;;
```

Arbres binaires entiers

En suivant cette description, on obtient le type ci-dessus, où l'on distingue les étiquettes des feuilles et des nœuds internes.

Arbres binaires entiers



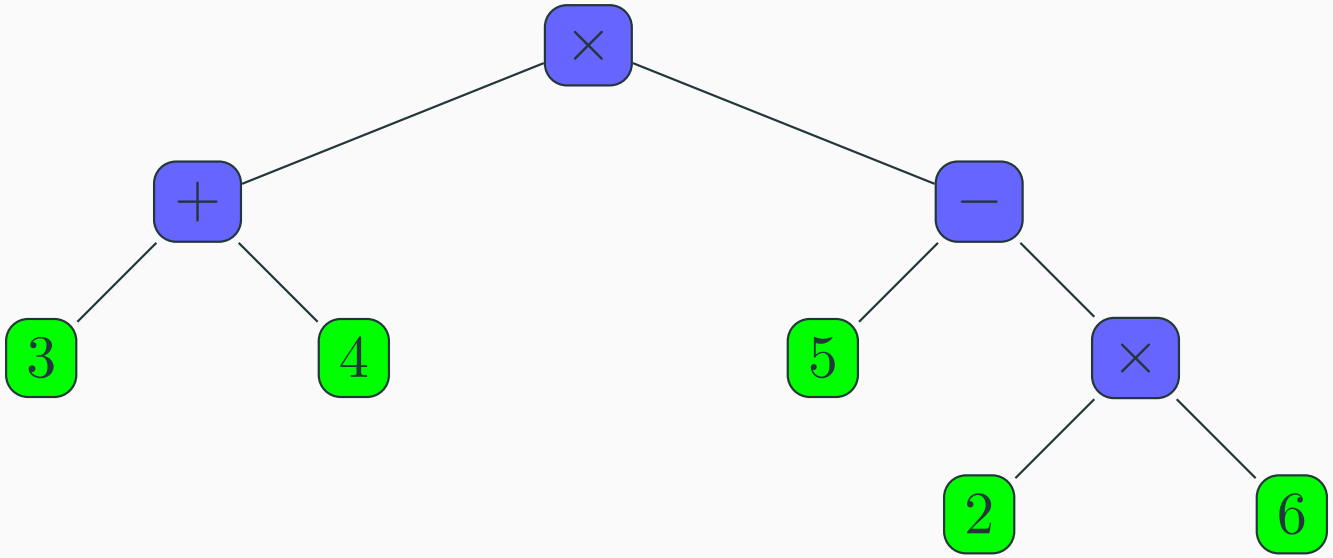
Exemple : Arbre syntaxique

Une expressions arithmétique se représente naturellement par un arbre binaire entier, les étiquettes des nœuds internes étant les opérateurs, et celles des feuilles étant les opérandes.

L'arbre ci-dessus représente l'expression

$$(3 + 4) \times (5 - (2 \times 6))$$

Arbres binaires entiers



Représentation en OCaml

```
1  type op = Plus | Moins | Fois | Div | Mod ;;
2  let expr = N (Fois, N (Plus, F 3, F 4), N (Moins, F 5, N (Fois, F 2, F 6))) ;;
```

OCaml

On peut utiliser un **type énuméré** pour définir les opérateurs.

Arbres binaires entiers

Évaluation

L'évaluation d'une telle expression se fait **récursivement**, par **filtrage**.

On écrit d'abord la fonction **`traduit`** qui renvoie la fonction `int -> int -> int` associée à un opérateur.

Arbres binaires entiers

Implémentation

```
let traduit = function
  | Plus -> (+)
  | Moins -> (-)
  | Fois -> ( * )
  | Div -> (/)
  | Mod -> (mod)
;;

let rec evalue e = match e with
  | F x -> x
  | N (op, a, b) -> traduit op (evalue a) (evalue b)
;;
```

Top-level

```
# traduit ;;
- : op -> int -> int -> int = <fun>
# evalue ;;
- : (int, op) arbre -> int = <fun>
# evalue expr ;;
- : int = -49
```

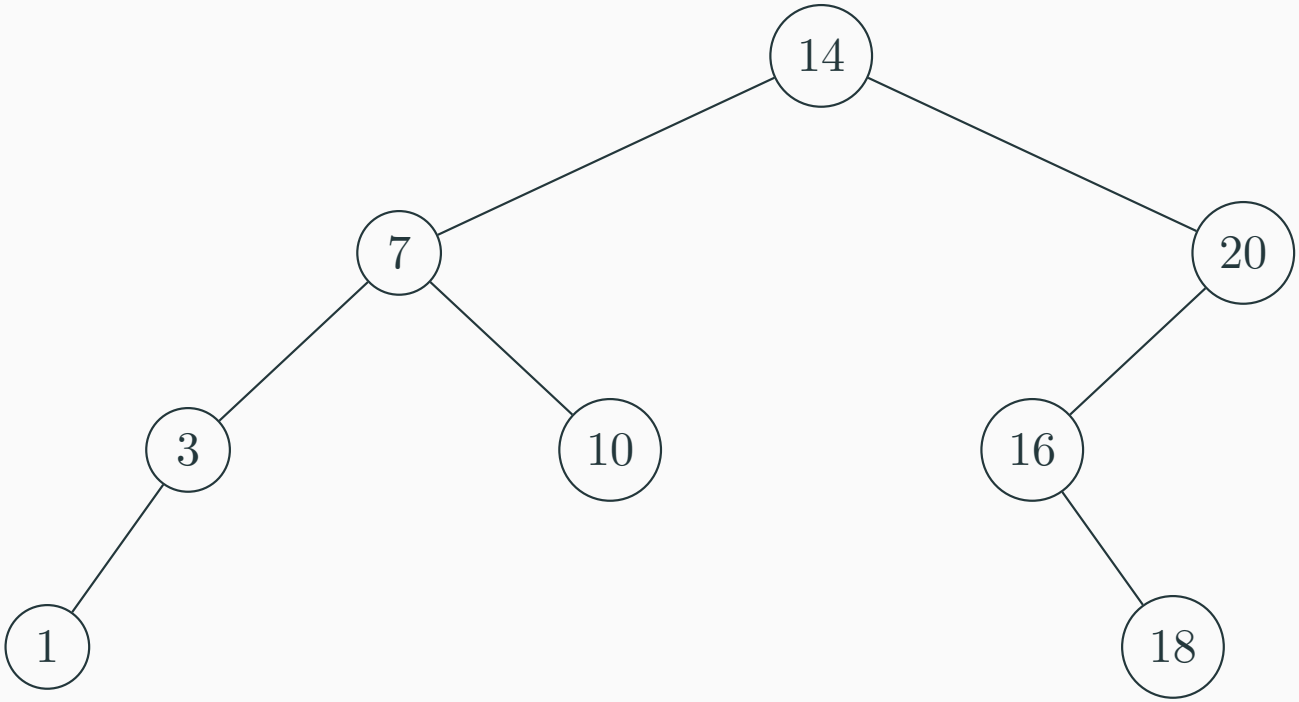
Arbres binaires

On propose maintenant une implémentation des arbres **binaires**. Elle est très proche de l'implémentation précédente des arbres **binaires entiers**.

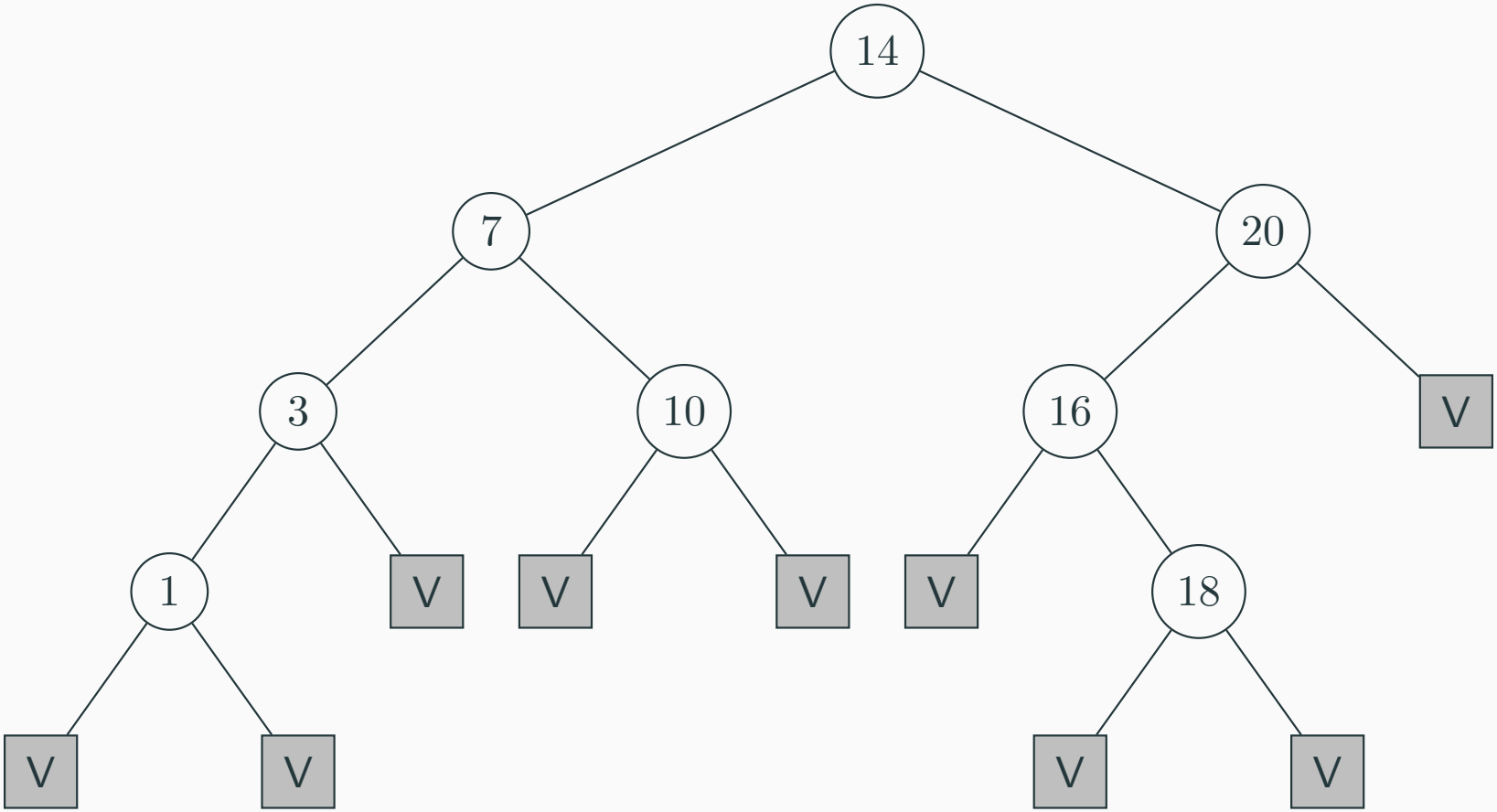
En effet, si on considère un arbre binaire, et qu'à chaque nœud x on fait pousser $2 - a$ feuilles où a est l'arité de x, on obtient un arbre binaire entier.

On note "Vide" les feuilles que l'ont fait pousser, de sorte que les feuilles du nouvel arbre sont toutes "Vides".

Arbres binaires



Arbres binaires



Arbres binaires

```
1  type 'a arbre = Vide
2                | N of 'a * 'a arbre * 'a arbre ;;
```

Implémentation

Avec cette représentation, un **arbre binaire** est :

- soit vide ;
- soit la donnée d'une étiquette, et deux sous-arbres binaires.

Ceci mène à la définition du type ci-dessus.

Arbres binaires

```
let ex_ab = N (14,
  N(7, N(3, N(1, Vide, Vide), Vide), N(10, Vide, Vide)),
  N(20, N(16, Vide, N(18, Vide, Vide)), Vide))
;;
```

Exemple

Voici l'implémentation de l'arbre de la figure précédente.

Arbres binaires

code

```
let rec hauteur a = match a with
  | Vide -> -1
  | N(_,g,d) -> 1 + max (hauteur g) (hauteur d)
;;
```

top-level

```
# hauteur ex_ab ;;
- : int = 3
```

Exemple

Voici une fonction permettant de calculer la hauteur d'un tel arbre.

Parcours d'arbres binaires entiers

Parcours d'arbres binaires entiers

Parcours d'arbre

On veut énumérer les nœuds d'un **arbre binaire entier**.

En pratique, on voudra faire divers traitements avec les étiquettes des nœuds.

À titre d'exemple, on va vouloir ici stocker l'énumération dans une liste, pour illustrer dans quel ordre on traite chaque nœud.

Parcours d'arbres binaires entiers

```
1  type ('a, 'b) etiq = A of 'a
2                     | B of 'b ;;
```

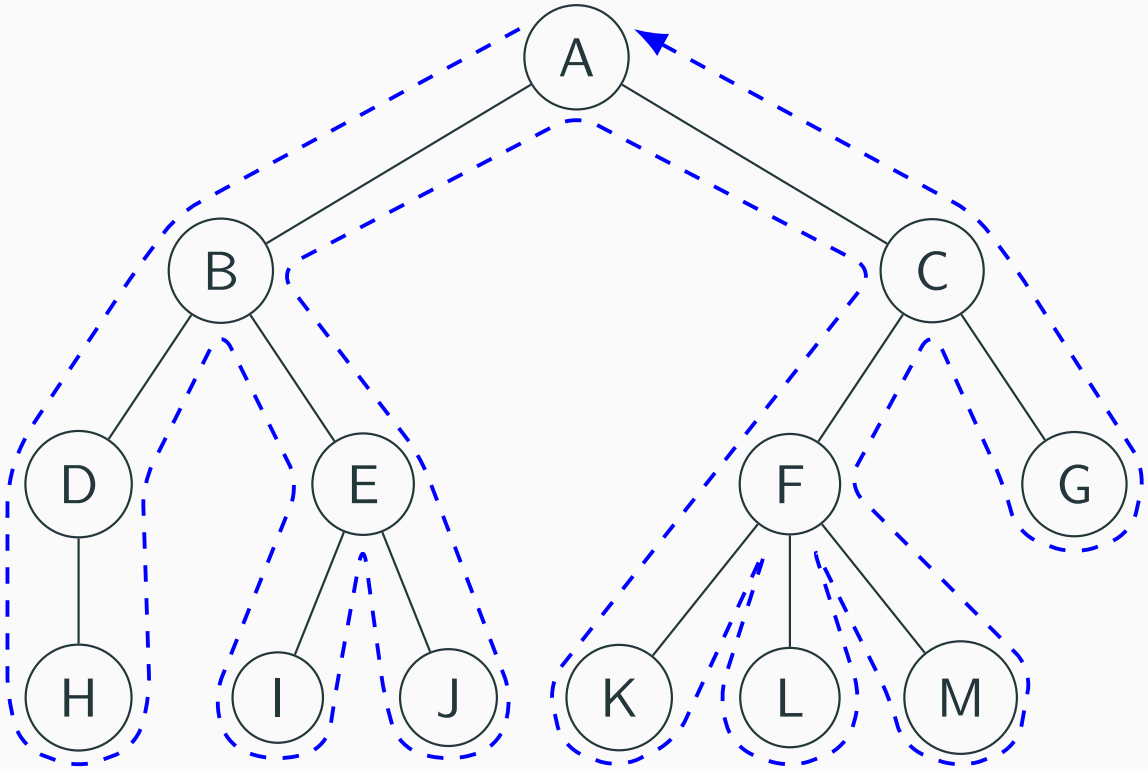
Homogénéité

Avant de commencer, il faut régler un léger problème : dans notre implémentation des arbres binaires entiers, on autorise les nœuds internes à avoir un type différent des feuilles.

Mais dans une liste, tous les éléments doivent avoir le même type !

Ainsi, on introduit un type pour stocker dans une même liste les étiquettes des nœuds internes et des feuilles.

Parcours en profondeur



Parcours en profondeur

Le **parcours en profondeur** d'un arbre consiste à descendre le plus possible dans l'arbre le long d'une branche avant de revenir en arrière explorer les autres branches.

Parcours en profondeur

Le **parcours en profondeur** d'un arbre consiste à descendre le plus possible dans l'arbre le long d'une branche avant de revenir en arrière explorer les autres branches.

Pour un **arbre binaire entier** de racine r, et de sous-arbres gauche et droit g et d, il y a trois choix naturels :

- racine r, énumération de g, énumération de d : on parle de **parcours préfixe**.
- énumération de g, racine r, énumération de d : on parle de **parcours infixe**.
- énumération de g, énumération de d, racine r : on parle de **parcours postfixe** (ou **suffixe**).

Parcours en profondeur

```
1   let rec prefixe a = match a with
2     | F x -> [B x]
3     | N(x,g,d) -> (A x)::(prefixe g)@(prefixe d)
4   ;;
5
6   let rec infixe a = match a with
7     | F x -> [B x]
8     | N(x,g,d) -> (infixe g)@((A x)::(infixe d))
9   ;;
10
11  let rec postfixe a = match a with
12    | F x -> [B x]
13    | N(x,g,d) -> ((postfixe g)@(postfixe d))@[A x]
14  ;;
```

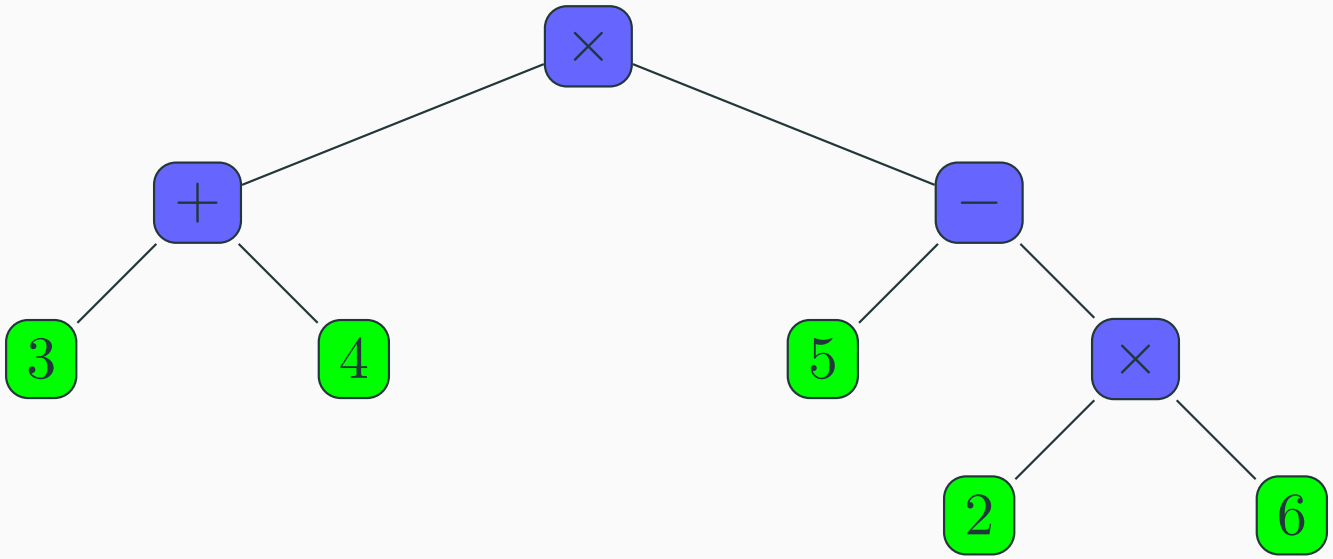
Parcours en profondeur

Voici l'écriture de ces trois parcours en profondeur.

Parcours en profondeur

(3+4)*(5-(2*6))

```
# prefixe expr ;;
- : (op, int) etiq list =
[A Fois; A Plus; B 3; B 4; A Moins; B 5; A Fois; B 2; B 6]
# infixe expr ;;
- : (op, int) etiq list =
[B 3; A Plus; B 4; A Fois; B 5; A Moins; B 2; A Fois; B 6]
# postfixe expr ;;
- : (op, int) etiq list =
[B 3; B 4; A Plus; B 5; B 2; B 6; A Fois; A Moins; A Fois]
```



Parcours en profondeur

Remarque

Il est notable de voir que l'énumération donnée par le parcours préfixe ou le parcours postfixe permet de reconstruire l'arbre, contrairement à celle du parcours infixe.

Parcours en profondeur

Proposition

Si les étiquettes des nœuds internes et des feuilles ont des types différents, alors à une énumération préfixe ou postfixe correspond un seul arbre binaire entier.

Preuve

On donne une preuve dans le cas de l'énumération postfixe.

L'argument est similaire pour l'énumération préfixe.

Parcours en profondeur

Lemme

Considérons l'énumération postfixe d'un arbre binaire entier. On parcourt l'énumération avec un compteur c initialisé à 0, on ajoute $+1$ pour une feuille, et -1 pour un nœud interne. Alors c est toujours strictement positif après le début de l'énumération, et vaut 1 à la fin.

Preuve du lemme

Par récurrence sur la longueur de l'énumération.

- Pour une énumération de longueur 1 (correspondant à une unique feuille), c'est immédiat.
- Sinon, l'énumération est constituée de
  - l'énumération du sous-arbre gauche (par HR, c est toujours > 0 et vaut 1 à la fin),
  - puis celle du sous-arbre droit (donc c reste > 1 et termine par 2),
  - et enfin la racine qui est un nœud interne, donc $s = 1$ à la fin.

Ainsi, par principe de récurrence, le lemme est démontré.

Parcours en profondeur

Preuve de la proposition

Par récurrence sur la longueur de l'énumération.

- une énumération de taille 1 correspond à un arbre à une unique feuille, l'unicité est donc évidente.
- Donnons nous maintenant une énumération postfixe d'un arbre binaire entier, de taille > 1, et supposons la propriété démontrée pour des énumérations plus petites.
  Par nature de l'énumération, se trouve d'abord toute l'énumération du sous-arbre gauche, puis celle du sous-arbre droit, et enfin la racine de l'arbre.

Preuve de la proposition

Pour pouvoir appliquer l'hypothèse de récurrence et terminer la preuve, il suffit de savoir où se situe la frontière entre les énumérations des sous-arbres gauche et droit.

Or, on déduit du lemme que cette frontière se situe juste après que le compteur c du lemme ait pris la valeur 1 pour la dernière fois (avant la racine) : on peut donc reconstruire l'arbre par hypothèse de récurrence.

Ainsi, par principe de récurrence, la propriété est démontré.

Parcours en profondeur

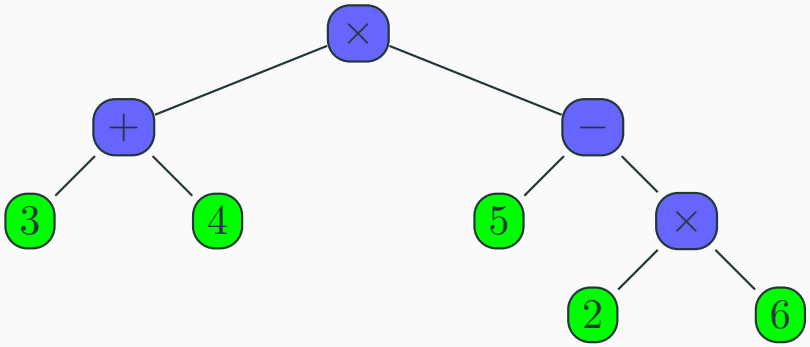
Remarque

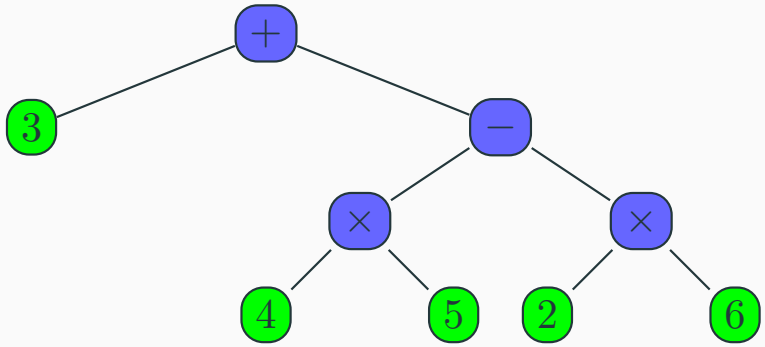
Cette propriété sur l'énumération postfixe a été utilisée dans certaines calculatrices, et s'étend à des expressions faisant usage d'opérateurs d'arité différente de 2. On parle de calculatrice à notation **polonaise inversée**.

L'intérêt est que les parenthèses sont inutiles pour donner l'expression arithmétique, ce qui fournit un gain de temps à l'utilisateur.

Une pile suffit pour écrire une fonction d'évaluation d'une expression donnée sous la forme du parcours postfixe, qu'on laisse en exercice (vous le ferez sans doute l'an prochain en IPT).

Parcours en profondeur





Exemple

L'énumération ne suffit pas pour reconstruire l'arbre, comme le montre les deux arbres ci-dessus qui ont la même énumération :

$$3 + 4 \times 5 - 2 \times 6$$

Parcours en largeur

Parcours en largeur

Contrairement au parcours en profondeur, le **parcours en largeur** liste tous les nœuds par profondeur croissante.

Par convention, les nœuds situés à gauche sont à énumérer en premier.

L'énumération est un peu plus délicate à obtenir qu'une énumération d'un parcours en profondeur.

Parcours en largeur

```
let rec racines l = match l with
  | [] -> []
  | (F x)::q -> (B x)::(racines q)
  | N(x,_,_)::q -> (A x)::(racines q)
;;
```

```
let rec ss_arbres l = match l with
  | [] -> []
  | (F _)::q -> ss_arbres q
  | N(_,g,d)::q -> g::d::ss_arbres q
;;
```

```
let largeur a =
  let rec aux l = match l with
    | [] -> []
    | _ -> (racines l)@(aux (ss_arbres l))
  in aux [a]
;;
```

Implémentation

On commence par écrire deux fonctions qui prennent en entrée une liste d'arbres (qu'on appelle une **forêt**), et renvoient respectivement l'énumération de leurs racines, et la liste de leurs sous-arbres.

Parcours en largeur

```
let rec racines l = match l with
  | [] -> []
  | (F x)::q -> (B x)::(racines q)
  | N(x,_,_)::q -> (A x)::(racines q)
;;
```

```
let rec ss_arbres l = match l with
  | [] -> []
  | (F _)::q -> ss_arbres q
  | N(_,g,d)::q -> g::d::ss_arbres q
;;
```

```
let largeur a =
  let rec aux l = match l with
    | [] -> []
    | _ -> (racines l)@(aux (ss_arbres l))
  in aux [a]
;;
```

Implémentation

La fonction aux de `largeur` renvoie l'énumération en largeur d'une liste d'arbres : si la liste est non vide, elle énumère les racines, et se rappelle récursivement sur la liste des sous-arbres.

Parcours en largeur

```
let rec racines l = match l with
  | [] -> []
  | (F x)::q -> (B x)::(racines q)
  | N(x,_,_)::q -> (A x)::(racines q)
;;
```

```
let rec ss_arbres l = match l with
  | [] -> []
  | (F _)::q -> ss_arbres q
  | N(_,g,d)::q -> g::d::ss_arbres q
;;
```

```
let largeur a =
  let rec aux l = match l with
    | [] -> []
    | _ -> (racines l)@(aux (ss_arbres l))
  in aux [a]
;;
```

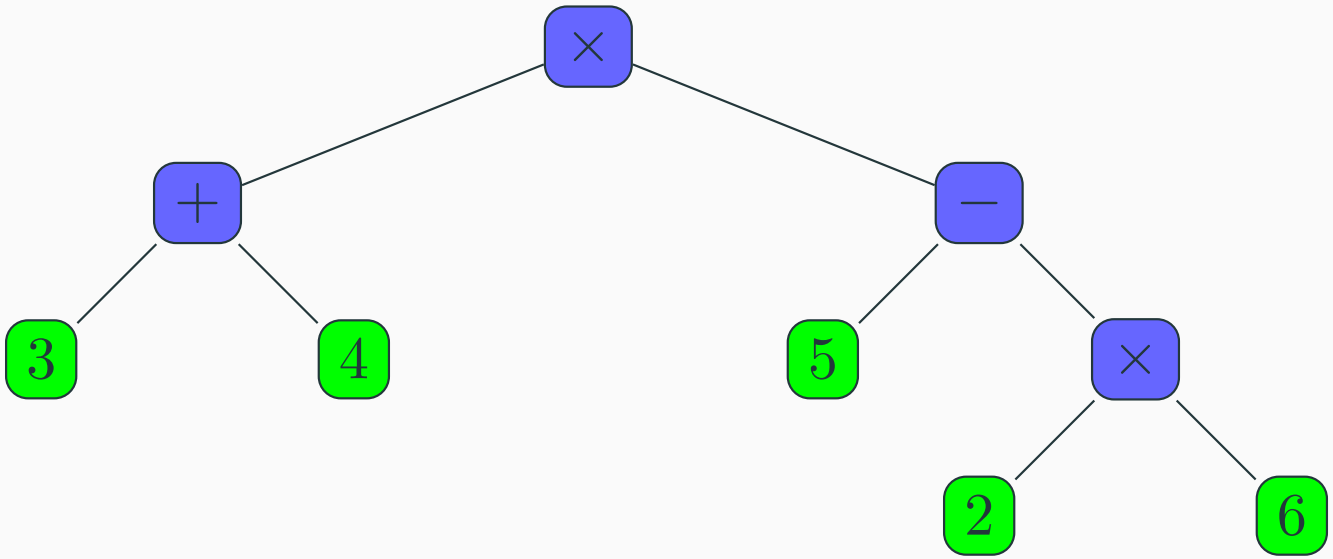
Implémentation

En pratique, lors du k-ème appel à aux, on travaille sur la liste des sous-arbres de a enracinés à un nœud de profondeur $k-1$.

Parcours en largeur

(3+4)*(5-(2*6))

```
#largeur expr ;;
- : (op, int) etiq list =
[A Fois; A Plus; A Moins; B 3; B 4; B 5; A Fois; B 2; B 6]
```



Parcours d'arbres

Remarques

- Les parcours en profondeur **préfixe** et **postfixe**, ainsi que le parcours en **largeur**, se généralisent à d'autres arbres que les arbres binaires entiers, avec des fonctions assez semblables à celles vues dans ce chapitre.
- Le parcours **infixe** se généralise à des arbres binaires (non nécessairement binaires entiers), mais pas à des arbres quelconques.

Une borne inférieure sur la complexité des algorithmes de tri

Borne inférieure sur la complexité des tris par comparaisons

Peut-on faire mieux ?

- Le tri rapide a une complexité **quasi-linéaire** en moyenne.
- Le tri fusion a une complexité **quasi-linéaire** même dans le cas pire.

La prochaine question naturelle est la suivante : peut-on faire mieux ?

On va montrer que **non**.

Borne inférieure sur la complexité des tris par comparaisons

Preuve

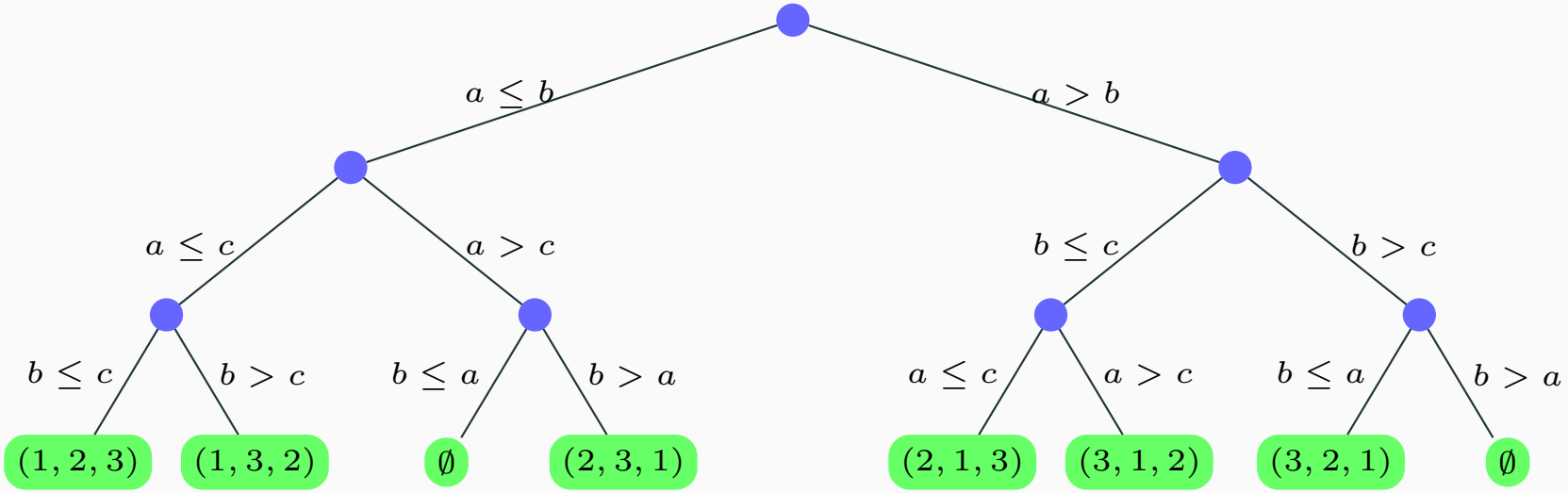
Considérons un algorithme de tri par comparaisons quelconque sur des listes de la forme $[\sigma(0), \sigma(1), \ldots, \sigma(n-1)]$, où $\sigma \in \mathfrak{S}_n$.

À chaque algorithme de tri, on peut associer un **arbre binaire** qui représente le tri selon les comparaisons effectuées.

- On part d'une liste $[\sigma(0), \sigma(1), \ldots, \sigma(n-1)]$, et on termine avec la liste $[0, 1, \ldots, n-1]$.
- Selon le résultat d'une comparaison entre $\sigma(i$ et $\sigma(j)$, une branche de l'arbre ou l'autre est suivie.
- Avec deux permutations distinctes, on se retrouve à des **feuilles** distinctes de l'arbre.

Ainsi, l'arbre dont on parle a au moins $|\mathfrak{S}_n| = n!$ feuilles.

Exemple



Exemple

L'arbre ci-dessus est celui associé au tri par sélection sur une liste de 3 éléments $[a, b, c]$.

Il a 8 feuilles : 6 d'entre elles sont associées à une permutation, les 2 autres correspondent au résultat de 2 comparaisons incompatibles avec une liste d'entrées distinctes.

Borne inférieure sur la complexité des tris par comparaisons

Preuve

Dans un tel arbre, l'exécution de l'algorithme correspondant est représenté par l'une de ses branches, et le nombre de comparaisons effectuées est donné par la **longueur** de cette branche.

La **complexité** de l'algorithme correspond donc à la **hauteur** de l'arbre.

Or un arbre binaire de hauteur h a au plus 2^h feuilles.

Donc un arbre binaire ayant au moins $n!$ feuilles a une hauteur d'au moins $\log_2(n!)$.

$$n! \sim_{+\infty} n^n e^{-n} \sqrt{2\pi n}$$

Donc

$$\log_2(n!) \sim_{+\infty} n \log_2(n)$$

Borne inférieure sur la complexité des tris par comparaisons

Borne inférieure

Ainsi, la complexité dans le pire cas d'un algorithme de tri par comparaisons quelconque a une complexité d'au moins $\Theta(n \log_2(n))$.