Les arbres

Option 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.
- $\forall x \in A \setminus \{r\}\ \exists n > 0,\ \exists (x_1, \ldots, x_n) \in A^n,$
  $x\ \mathcal{R}\ x_1\ \mathcal{R}\ x_2\ \mathcal{R} \cdots \mathcal{R}\ x_n = r$.

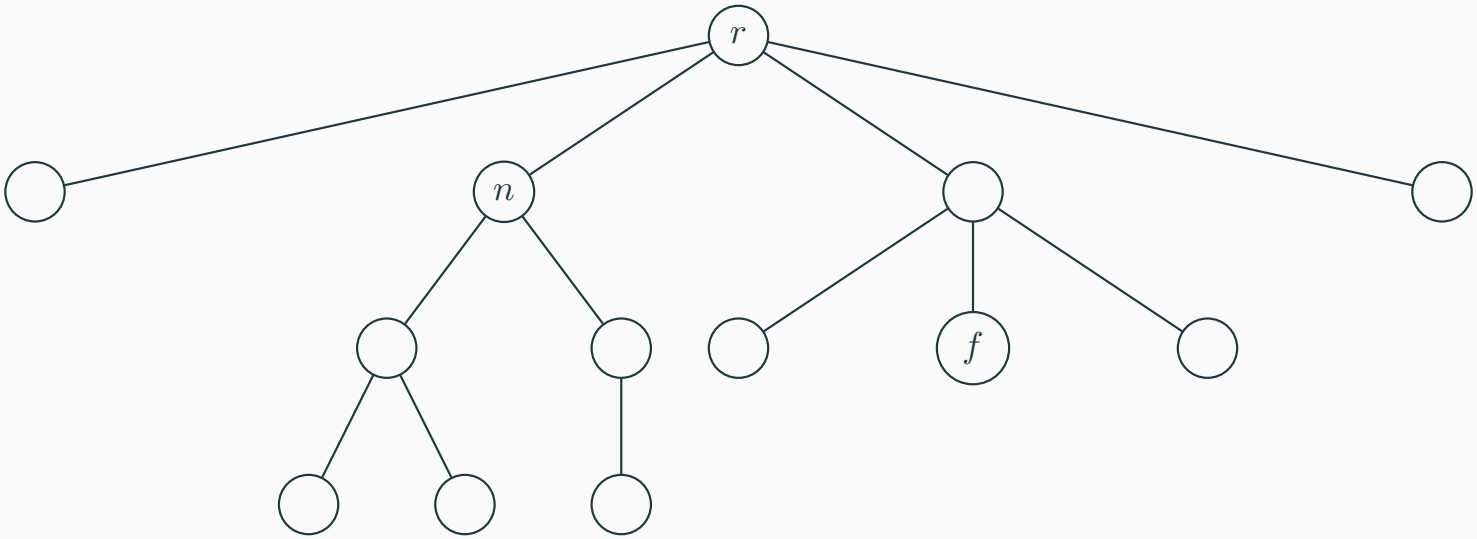
Définitions

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.

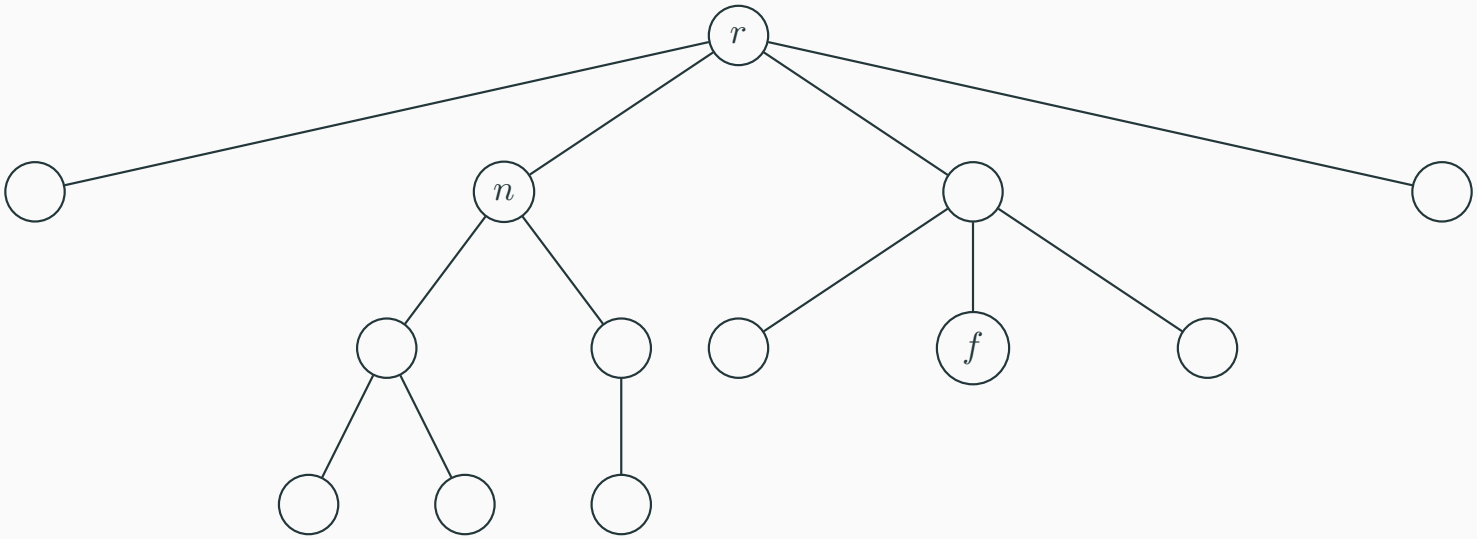
Définitions



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)

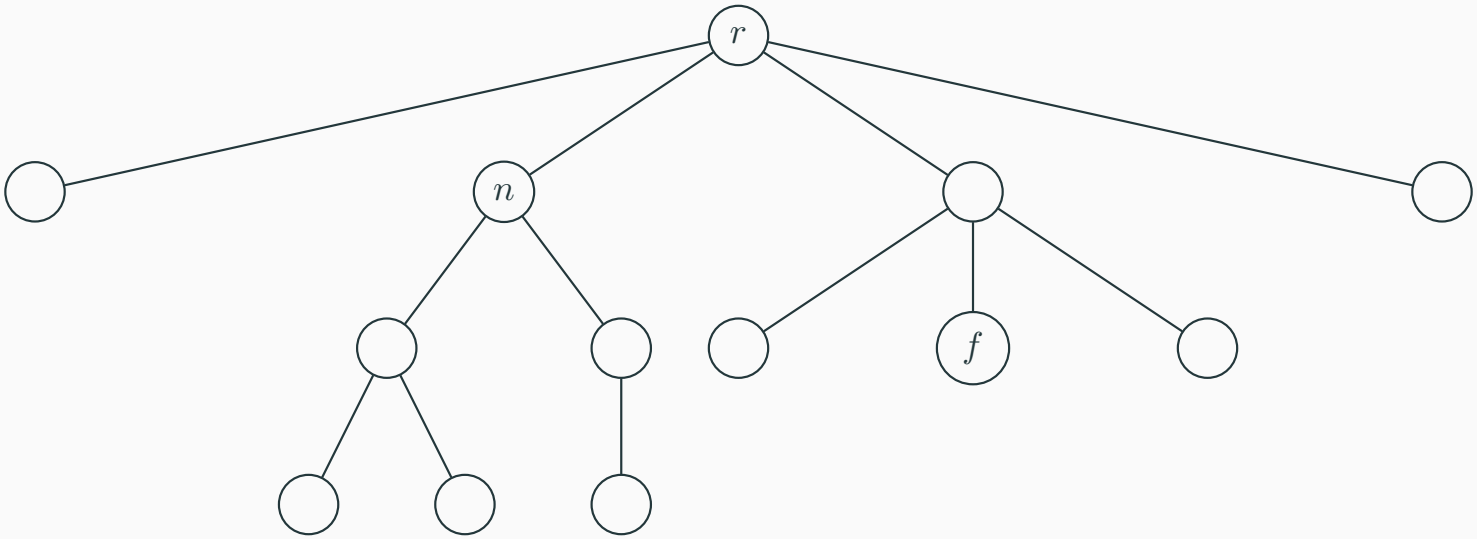
Les éléments de A sont appelés les **nœuds** de l'arbre.

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.

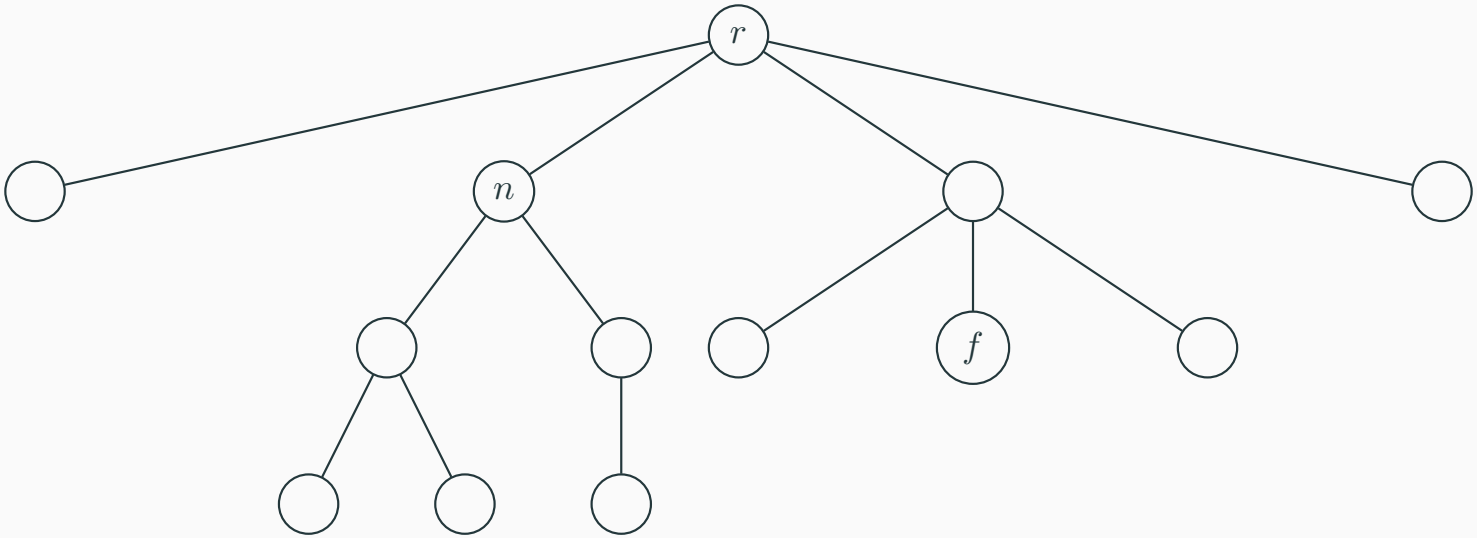
Définitions

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



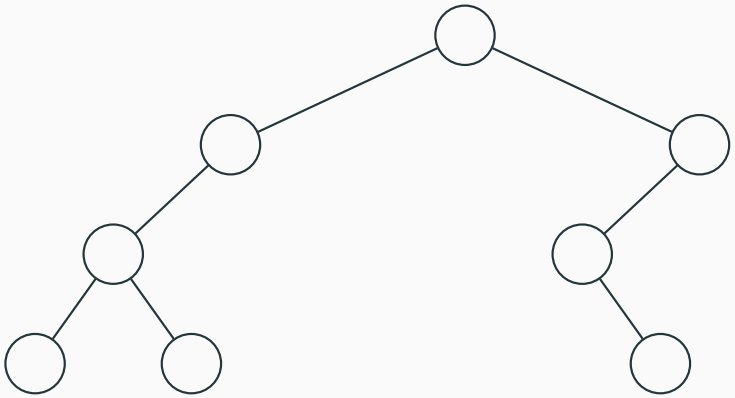
Exemple

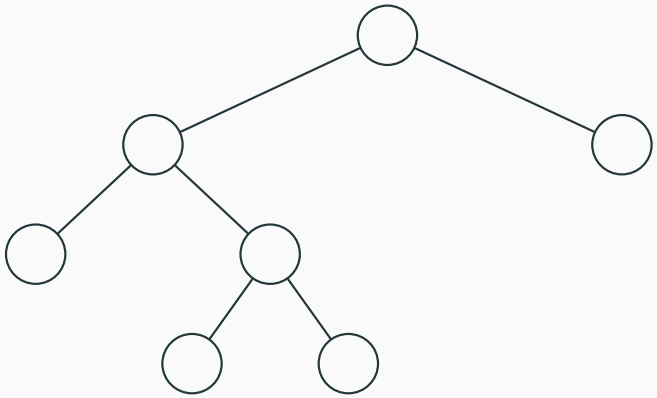
La racine est l'unique nœud à profondeur 0.

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.

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).

Exemple : XML

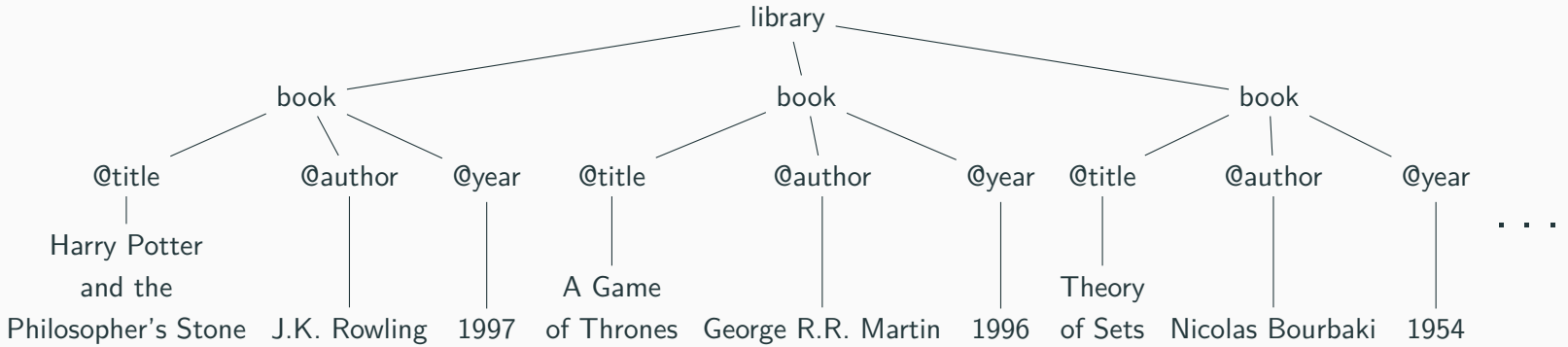
Le langage XML (eXtended Markup Language), créé à la fin des années 90, permet de représenter des données de manière structurée.

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 ;;
```

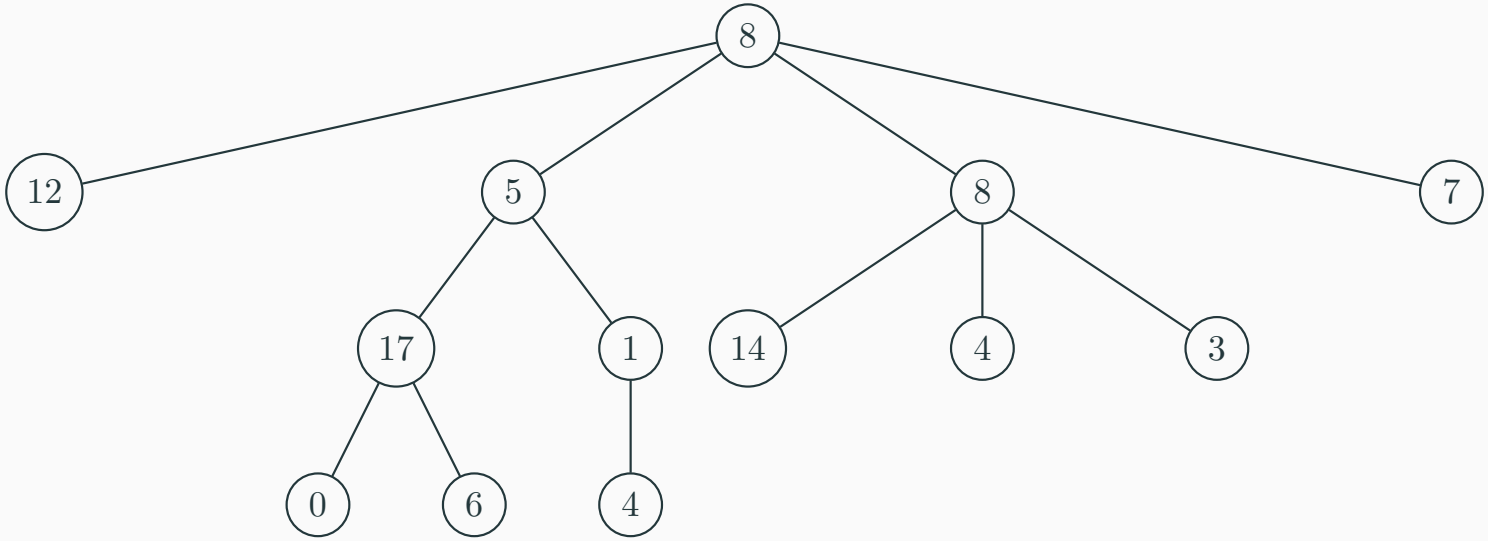
Arbres généraux

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).

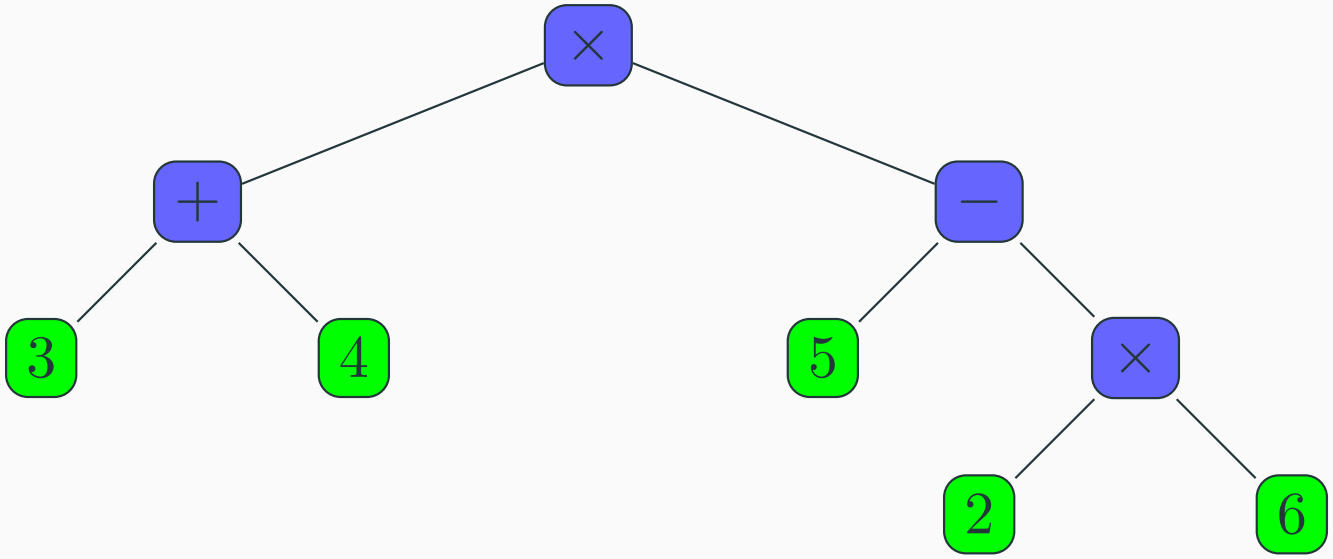
Arbres binaires entiers

```
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



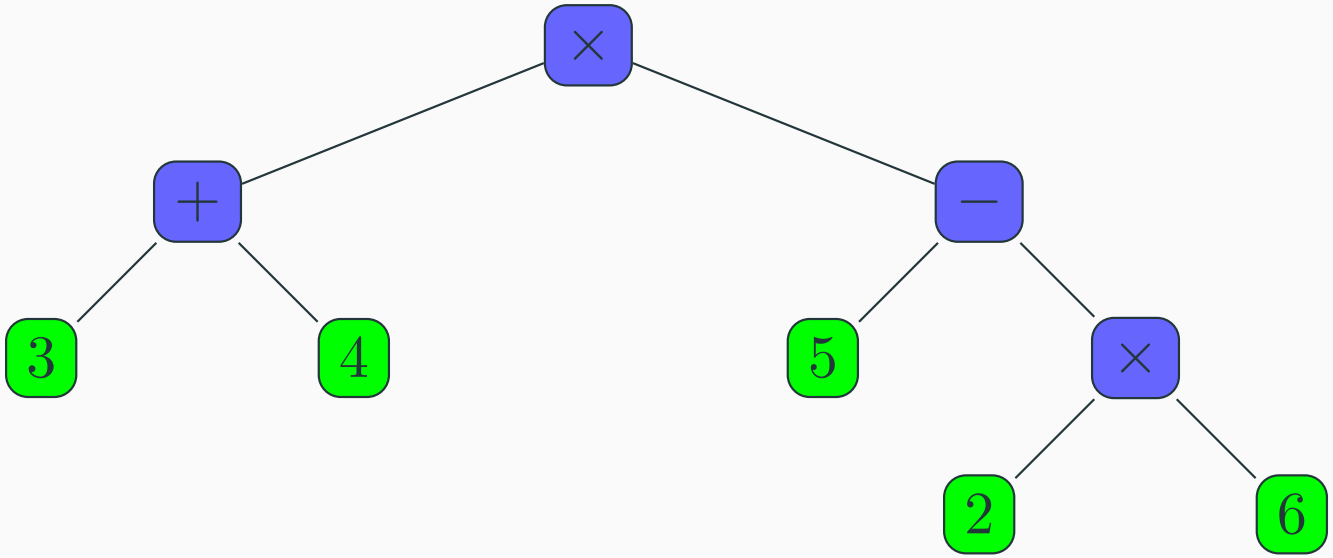
Exemple : Arbre syntaxique

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
```

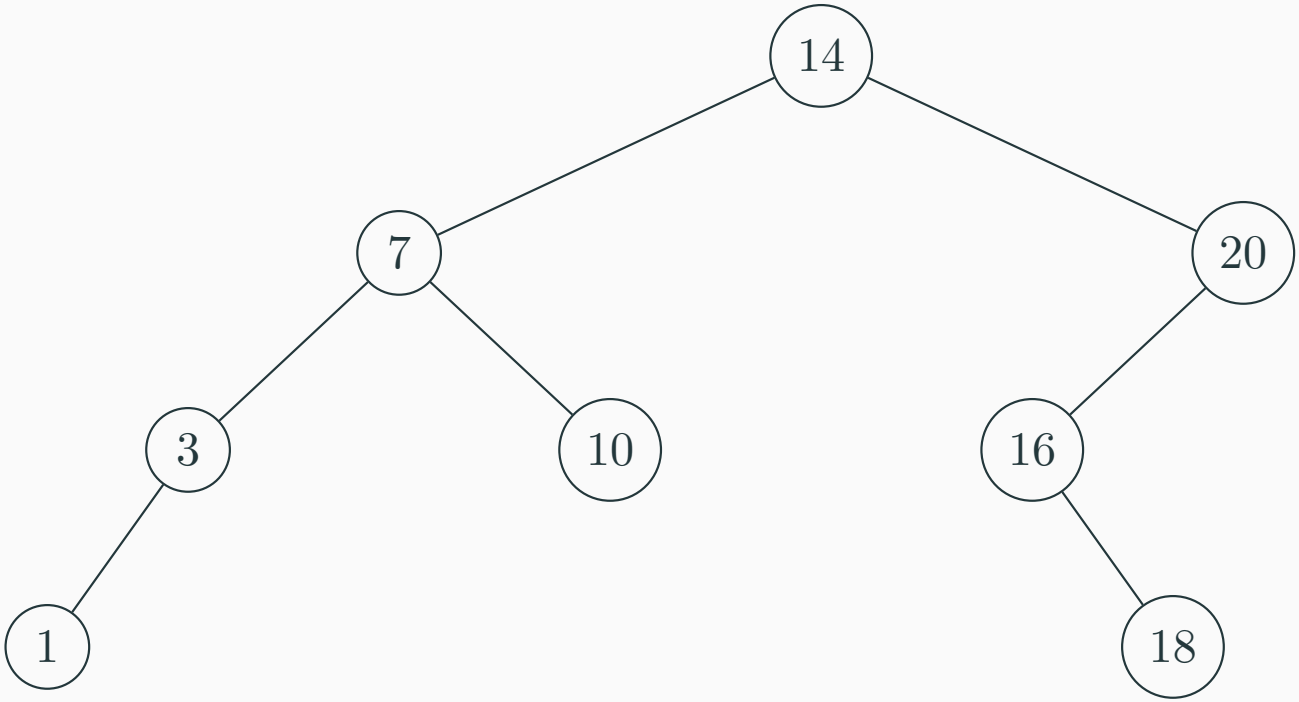
Arbres binaires

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

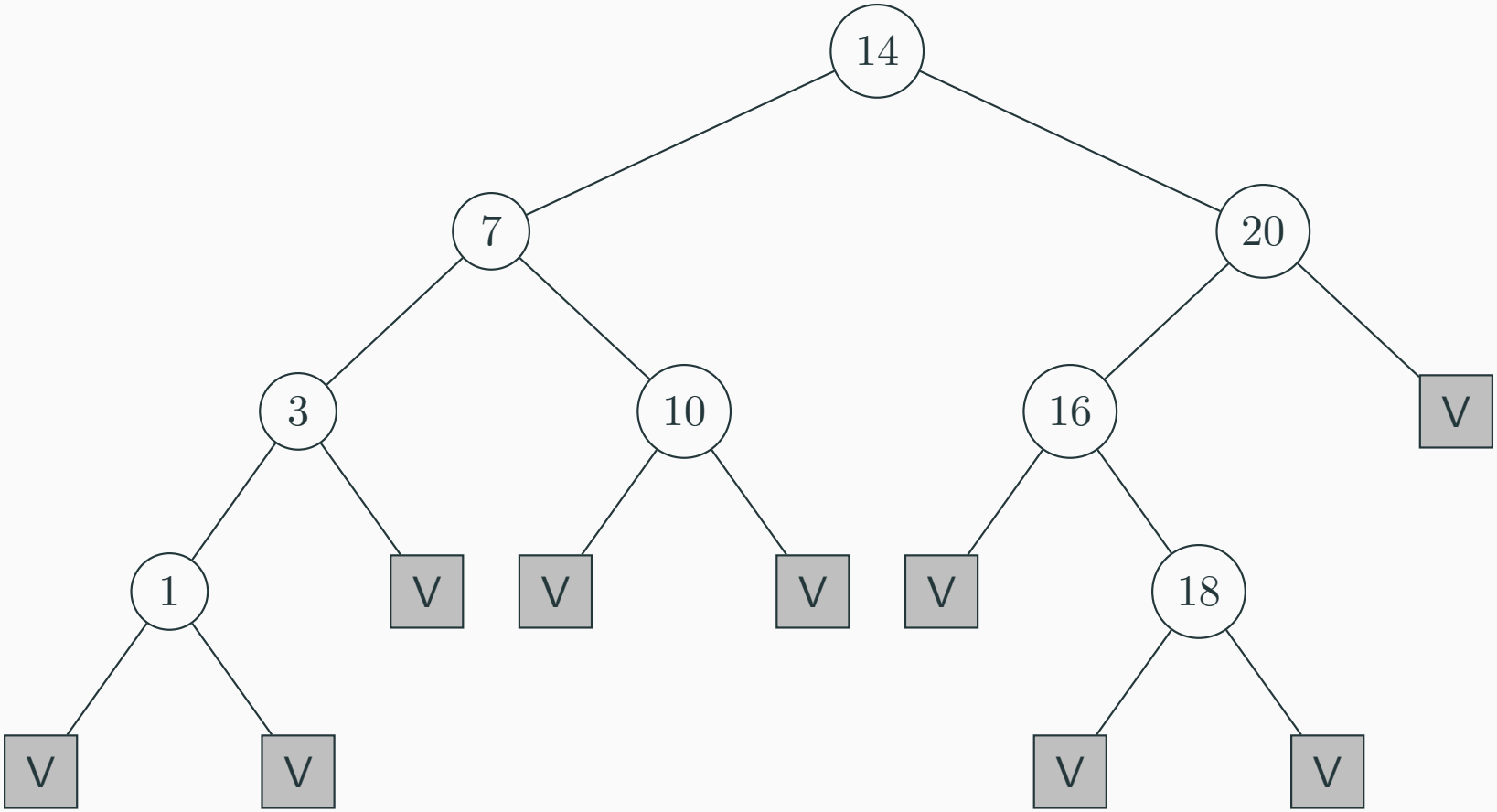
En effet, si on considère un arbre binaire, et qu'à chaque nœud x on fait pousser $2 - a$ feuilles où a est l'arité de x, on obtient un arbre binaire entier.

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 ;;
```

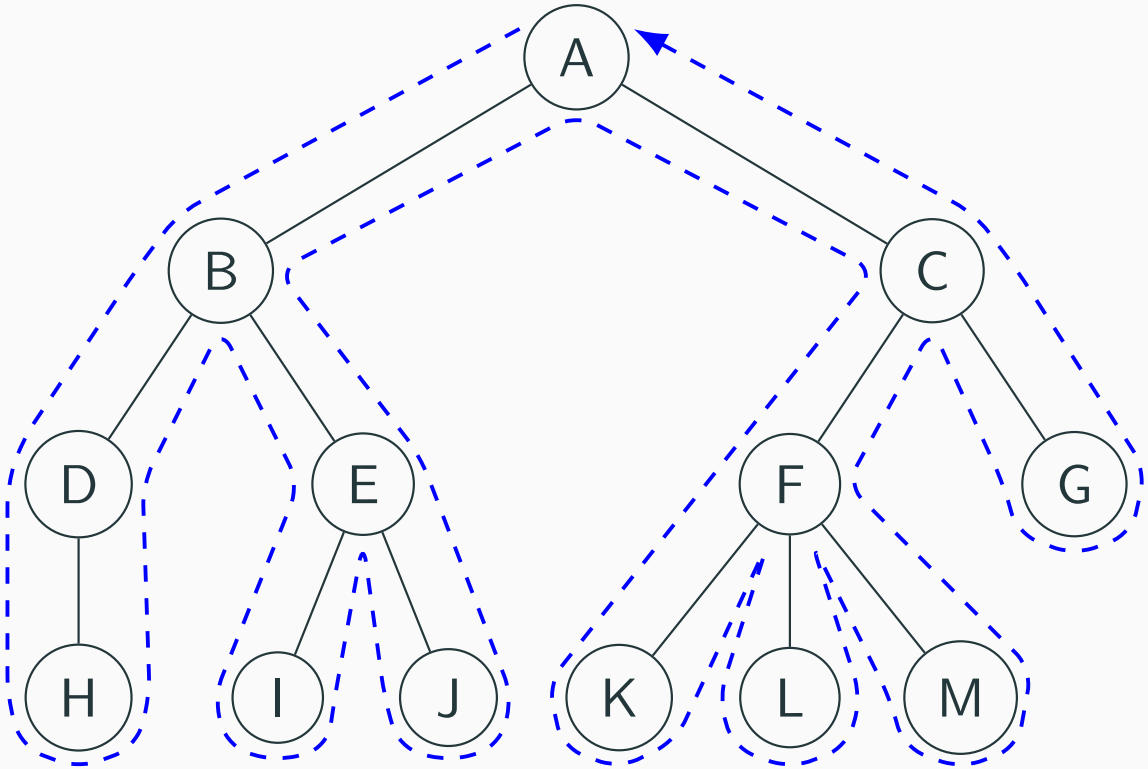
Homogénéité

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  ;;
```

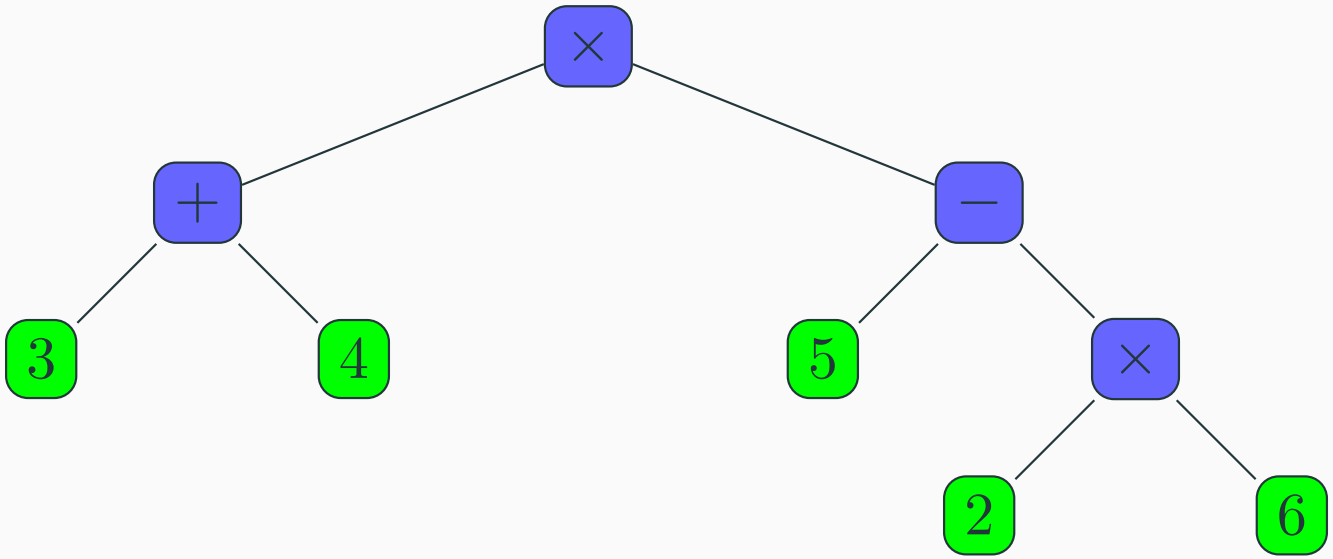
Parcours en profondeur

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

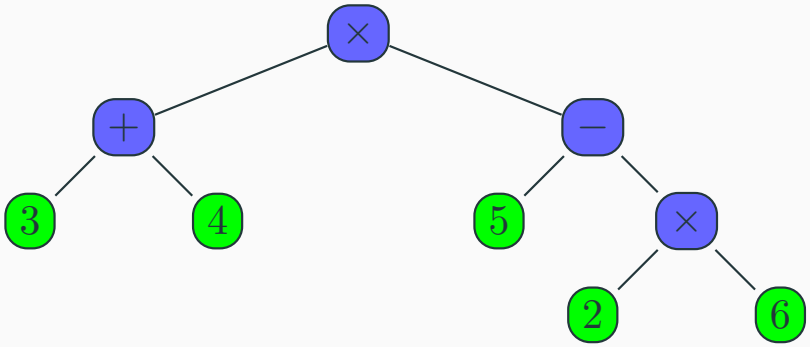
Remarque

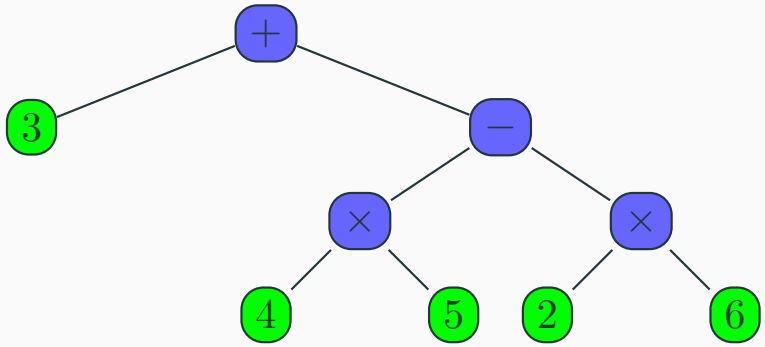
Cette propriété sur l'énumération postfixe a été utilisée dans certaines calculatrices, et s'étend à des expressions faisant usage d'opérateurs d'arité différente de 2. On parle de calculatrice à notation **polonaise inversée**.

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]
;;
```

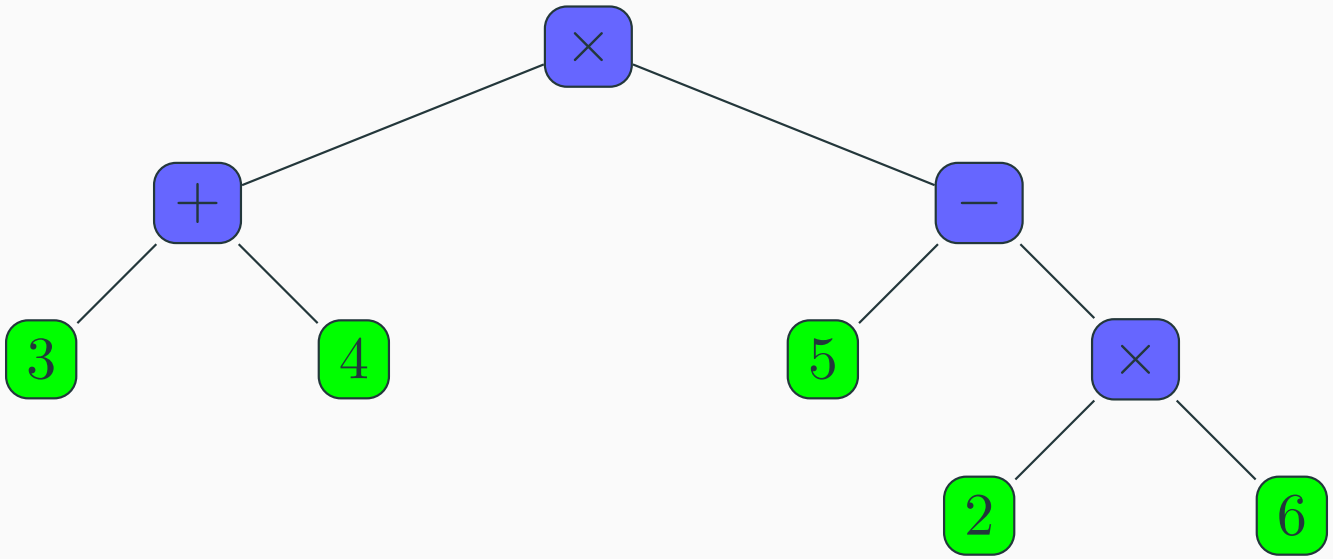
Implémentation

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.

Arbres binaires de recherche

Implémentation d'un dictionnaire avec une liste chaînée

Liste chaînée

Utiliser une structure de liste chaînée pour implémenter une structure de dictionnaire n'est pas efficace du point de vue de la complexité ; il faut stocker des couples (clé, élément) où les clés sont des éléments distincts, et rechercher si une clé est présente se fait en $O(n)$ dans le pire des cas.

On va maintenant proposer une structure efficace à l'aide d'arbres, plus précisément d'**arbres binaires de recherche**.

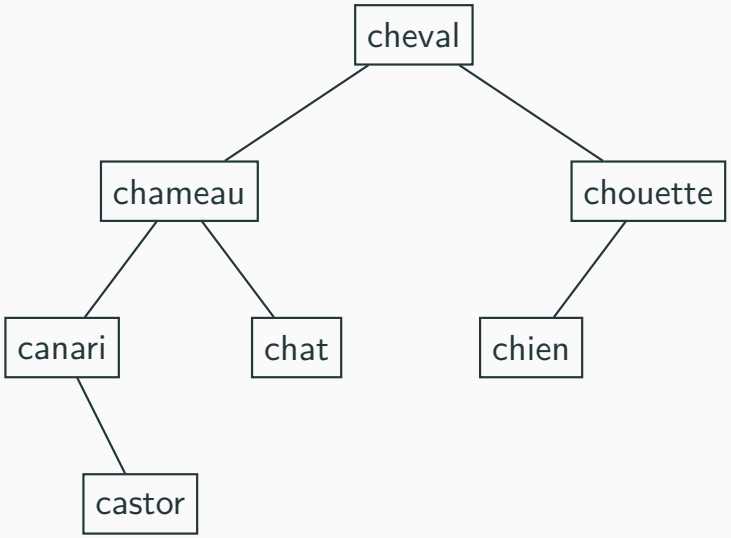
Structure d'arbre binaire de recherche

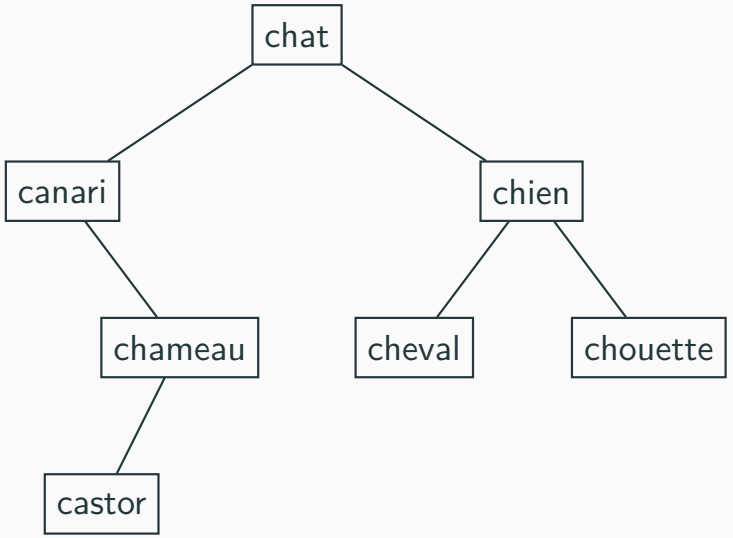
Définition (Arbre binaire de recherche)

Un **arbre binaire de recherche** (ABR) est un arbre binaire éventuellement vide dont les nœuds sont des couples (clé, valeur), tel que :

- deux nœuds n'ont **jamais** la **même clé** ;
- les clés sont à valeur dans un ensemble **totalement ordonné** ;
- pour tout nœud de l'arbre, les clés du **sous-arbre gauche** sont **strictement inférieures** à celle du nœud, et les clés du **sous-arbre droit** sont **strictement supérieures** à celle du nœud.

Structure d'arbre binaire de recherche





Exemple

Voici deux exemples d'ABR dont l'ensemble des clés est : {canari, castor, chameau, chat, cheval, chien, chouette}.

L'ordre utilisé est l'ordre alphabétique.

On n'a pas mis les valeurs associées, qui pourraient être les définitions de ces mots dans la langue française.

Structure d'arbre binaire de recherche

Proposition

Un arbre binaire est un **ABR** si et seulement si l'**énumération infixe** de ses nœuds est **strictement croissante**.

Remarque

L'**énumération infixe** n'est pas unique : elle correspond à plusieurs **ABR**.

C'est le cas par exemple pour les ABR précédents : les clés sont les mêmes donc les deux arbres ont la même **énumération infixe**.

Structure d'arbre binaire de recherche

Définition (Hauteur)

La **hauteur** d'un ABR est définie inductivement comme suit :

- la hauteur de l'arbre vide est -1 ;
- pour un arbre non vide, elle est égale au maximum des hauteurs de ses sous-arbres gauche et droit, $+\ 1$.

Remarque

On peut considérer qu'un ABR non vide est un arbre binaire **entier**, les "**feuilles**" de l'arbre étant des arbres **vides**.

C'est cohérent avec la définition de la hauteur, et pratique avec l'implémentation OCaml que l'on va voir dans ce chapitre.

Structure d'arbre binaire de recherche

Proposition

Le nombre de nœuds n d'un ABR (non vide) de hauteur h vérifie : $h + 1 \leq n \leq 2^{h+1} - 1$.

Preuve

La borne $2^{h+1} - 1$ est obtenue pour un arbre binaire **complet**, comme dans le cas des tas.

Un **chemin** de la racine a une feuille de profondeur h comporte $h + 1$ nœuds, ce qui démontre l'autre borne.

Structure d'arbre binaire de recherche

Corollaire

La hauteur h d'un ABR (non vide) à n nœuds vérifie :
$\lfloor \log_2(n) \rfloor \leq h \leq n - 1$.

Preuve

En passant au $\log$ dans l'encadrement précédent, on obtient :
$\log_2(n+1) - 1 \leq h \leq n - 1$.

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

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

Implémentation des ABR en Ocaml

Implémentation persistante

On choisit une implémentation **persistante** des **ABR**, identique à celle classique des arbres binaires.

Contrairement à l'implémentation des tas de la section précédente, les fonctions qui travailleront sur les arbres ne modifieront pas la structure mais renverront de nouveaux arbres.

En pratique, on implémente les ABR comme des **arbres binaires entiers** (tout nœud interne a 2 fils), les feuilles correspondent à un **arbre vide**.

Implémentation des ABR en Ocaml

```
1  type 'a abr = Vide | N of 'a abr * 'a * 'a abr;;
2
3  let rec hauteur a = match a with
4    | Vide -> -1
5    | N(g,_,d) -> 1 + max (hauteur g) (hauteur d)
6  ;;
```

Exemple

On peut facilement calculer la hauteur d'un tel arbre.

Implémentation de la structure de dictionnaire avec des ABR

```
1  type ('a, 'b) noeud = {cle : 'a; mutable valeur: 'b};;
```

Nœuds d'un ABR

En OCaml, on pourrait représenter les nœuds comme ci-dessus.

On a rendu le deuxième champ **mutable**, car dans un **dictionnaire**, on peut vouloir changer la valeur associée à une clé : dans ce cas, il suffit de retrouver le nœud à partir de la clé, et modifier la valeur associée.

Ainsi, la modification d'une valeur est essentiellement une recherche de clé.

Dans la suite, on oubliera les valeurs pour ne considérer que les clés, d'un point de vue algorithmique les opérations sont les mêmes.

Implémentation de la structure de dictionnaire avec des ABR

```
1  let creer_abr () = Vide ;;
2  let est_vide_abr a = a = Vide ;;
```

Fonctions basiques

Créer un ABR vide, et tester si un ABR est vide est facile.

Implémentation de la structure de dictionnaire avec des ABR

```
1  let rec rechercher a x = match a with
2    | Vide -> false
3    | N(g,y,_) when y>x -> rechercher g x
4    | N(_,y,d) when y<x -> rechercher d x
5    | _ -> true
6  ;;
```

Recherche

Si l'on **cherche** x dans un ABR de racine y, on est dans l'un des 3 cas suivants :

- soit $y = x$, auquel cas on a trouvé x ;
- soit $y < x$, auquel cas x ne peut se trouver que dans le sous-arbre droit du nœud étiqueté par y ;
- soit $y > x$, auquel cas x ne peut se trouver que dans le sous-arbre gauche du nœud étiqueté par y.

Implémentation de la structure de dictionnaire avec des ABR

```
1  let rec inserer a x = match a with
2    | Vide -> N(Vide,x,Vide)
3    | N(g,y,d) when y>x -> N(inserer g x, y, d)
4    | N(g,y,d) when y<x -> N(g, y, inserer d x)
5    | _ -> a
6  ;;
```

Insertion

Pour l'**insertion** de x dans l'ABR, on chemine dans l'arbre jusqu'à arriver à une feuille **`Vide`**, que l'on remplace par un arbre contenant simplement x.

On choisit de renvoyer l'arbre à l'identique si x est déjà présent.

Implémentation de la structure de dictionnaire avec des ABR

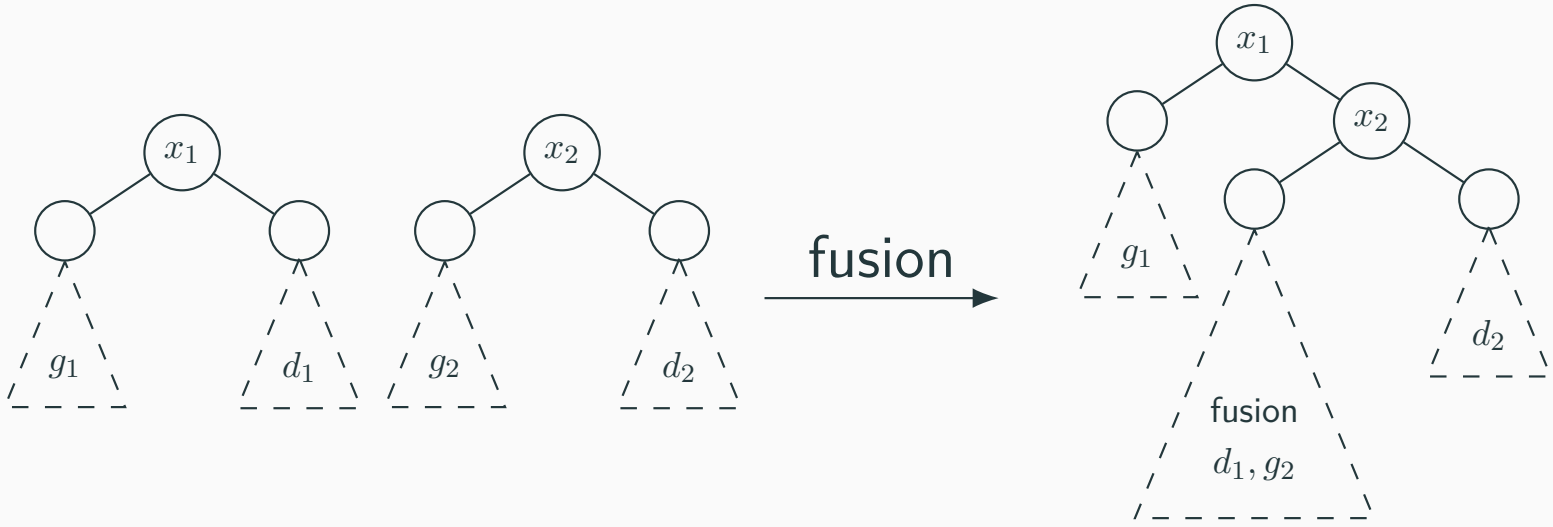
Suppression d'un élément dans un ABR

La **suppression** d'un élément est légèrement plus complexe.

Comme pour l'insertion, on va **descendre** dans l'arbre jusqu'à tomber sur l'élément en question, et on sera ramené à la suppression de la **racine** d'un ABR.

On propose alors deux solutions.

Implémentation de la structure de dictionnaire avec des ABR



Suppression par fusion

Si x est la racine de l'ABR et l'élément à supprimer, on peut **fusionner** ses deux sous-arbres gauche et droit.

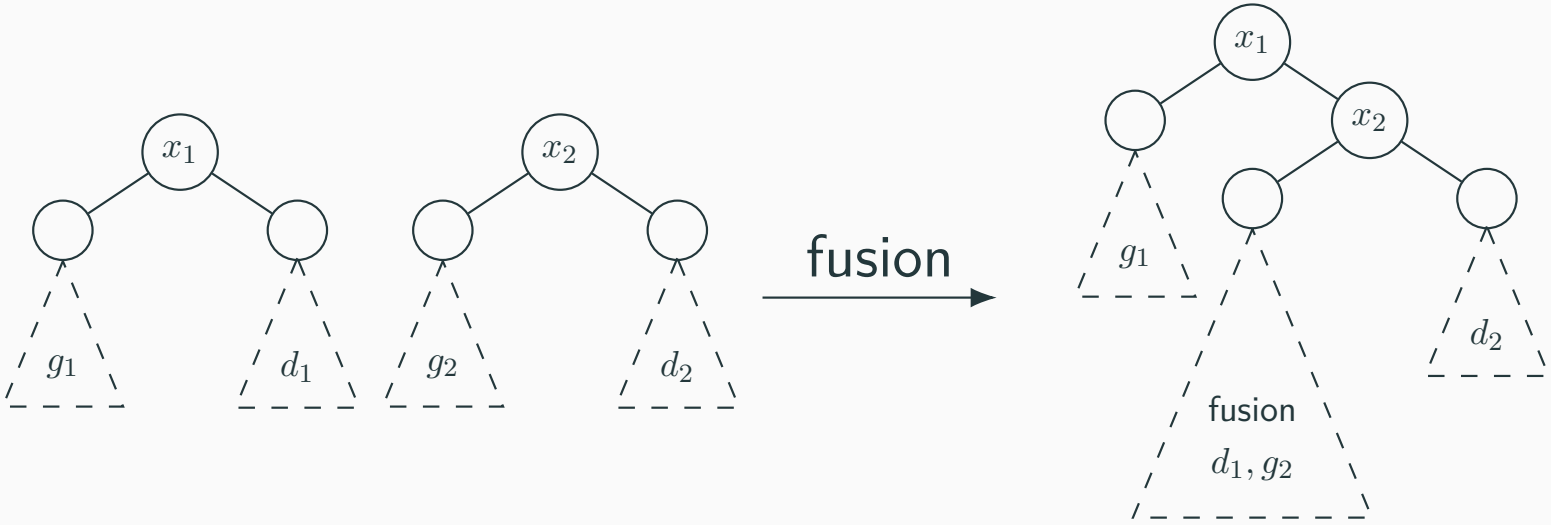
- Cette fusion est immédiate si l'un des deux sous-arbres est vide.
- Sinon, elle peut se faire selon le schéma ci-dessus, où x_1 et x_2 sont les fils de x.

Implémentation de la structure de dictionnaire avec des ABR

Remarques

- Cette méthode donne bien un ABR si les deux arbres concernés sont bien des ABR et que les clés du premier sont strictement inférieures aux clés du second.
- Le choix de x_1 comme nouvelle racine est arbitraire, et on pourrait choisir symétriquement x_2.
- De même que pour l'insertion, si le nœud n'est pas présent dans l'arbre, celui-ci est renvoyé à l'identique.

Implémentation de la structure de dictionnaire avec des ABR




```
1   let rec fusion a1 a2 = match a1,a2 with
2     | Vide,_ -> a2
3     | _,Vide -> a1
4     | N(g1,x1,d1), N(g2,x2,d2) -> N(g1,x1,N(fusion d1 g2,x2,d2))
5   ;;
6
7   let rec supprimer a x = match a with
8     | Vide -> Vide
9     | N(g,y,d) when y>x -> N(supprimer g x, y, d)
10    | N(g,y,d) when y<x -> N(g, y, supprimer d x)
11    | N(g,y,d) -> fusion g d
12  ;;
```

Implémentation de la structure de dictionnaire avec des ABR

```
1  let rec max_abr a = match a with
2    | Vide -> failwith "vide"
3    | N(_,x,Vide) -> x
4    | N(_,_,d) -> max_abr d
5  ;;
```

```
1  let rec supprimer a x = match a with
2    | Vide -> Vide
3    | N(g,y,d) when y>x -> N(supprimer g x, y, d)
4    | N(g,y,d) when y<x -> N(g, y, supprimer d x)
5    | N(Vide,y,d) -> d
6    | N(g,y,d) -> let z=max_abr g in N(supprimer g z, z, d)
7  ;;
```

Suppression du maximum du sous-arbre gauche

Supposons encore que la racine x soit l'élément à **supprimer**.

- Si le sous-arbre gauche est **vide**, l'arbre obtenu après suppression est simplement le sous-arbre droit.
- Sinon, on peut remplacer la racine par le maximum de son sous-arbre gauche (préalablement supprimé).

$\hookrightarrow$ On maintient ainsi bien la structure d'ABR.

Pour trouver le **maximum** d'un **ABR**, il suffit de descendre le plus à **droite** possible.

Implémentation de la structure de dictionnaire avec des ABR

Complexité

Toutes les opérations d'insertion, de suppression, et de recherche prennent un temps $O(h)$, où h est la **hauteur** de l'arbre.

Malheureusement, h peut être proche du nombre de nœuds, ce qui est mauvais en terme de complexité.

Exemple

En partant d'un arbre vide, si on insère successivement n nœuds dans l'ordre croissant, on obtient un arbre de hauteur $n-1$, et la construction se fait en temps $O(n^2)$ (on construit un "**peigne**").

Implémentation de la structure de dictionnaire avec des ABR

Solutions

Il y a plusieurs solutions pour remédier à ce problème.

- Si on se donne un ensemble de clés que l'on insère dans un ordre aléatoire (avec distribution uniforme sur les $n!$ permutations possibles), on peut montrer que l'espérance de la hauteur de l'arbre obtenu est $O(\log n)$.
  $\hookrightarrow$ En pratique, les choses se passent bien si on laisse faire le hasard.
- On peut rajouter de l'information dans l'ABR. Ces informations permettent, en utilisant des "rotations" bien choisies, d'équilibrer l'arbre pour garder la hauteur $O(\log n)$.

La suite du chapitre est dévolue à l'étude des arbres **AVL**, qui s'inscrivent dans la deuxième stratégie.

Arbres AVL

Définition

Un arbre **AVL** est un ABR tel que pour tout nœud, ses sous-arbres gauche et droit aient la même hauteur, à 1 près.

Remarque

Cette condition est suffisante pour que l'arbre soit approximativement équilibré.

Le nom des arbres **AVL** vient du nom de ses inventeurs : Georgii Adelson-Velsky et Evguenii Landis.

Théorème

La hauteur h d'un arbre **AVL** à n nœuds vérifie $h = O(\log n)$.

Preuve

Il suffit de voir qu'un arbre **AVL** a toujours un grand nombre de nœuds, relativement à sa hauteur.

Notons N_h le nombre de nœuds minimal d'un arbre **AVL** de hauteur h.

On a $N_0 = 1, N_1 = 2$, et $H_h = N_{h-1} + N_{h-2} + 1$. La suite (N_h+1) vérifie donc la même relation que la suite de Fibonacci.

Théorème

La hauteur h d'un arbre **AVL** à n nœuds vérifie $h = O(\log n)$.

Preuve

Ainsi, on peut montrer que $N_h = C\varphi^h + D(\frac{-1}{\varphi})^h - 1$, avec $\varphi = \frac{1+\sqrt{5}}{2}$, et C et D deux constantes. Donc, asymptotiquement, $N_h \sim C\varphi^h$.

Par suite, un arbre **AVL** à n nœuds possède une hauteur majorée par $\log_\varphi(n)$ (à une constante additive près), d'où le résultat.

Remarque

$\frac{1}{\log_2(\varphi)} \approx 1.44$, donc asymptotiquement un arbre **AVL** à n nœuds a une hauteur majorée par une quantité de l'ordre de $1.44 \times \log_2(n)$.

On n'est pas très loin des arbres binaires **complets** pour lesquels la hauteur est bornée par $\log_2(n)$.

Rotations et maintien de la structure d'arbre AVL

Arbres AVL

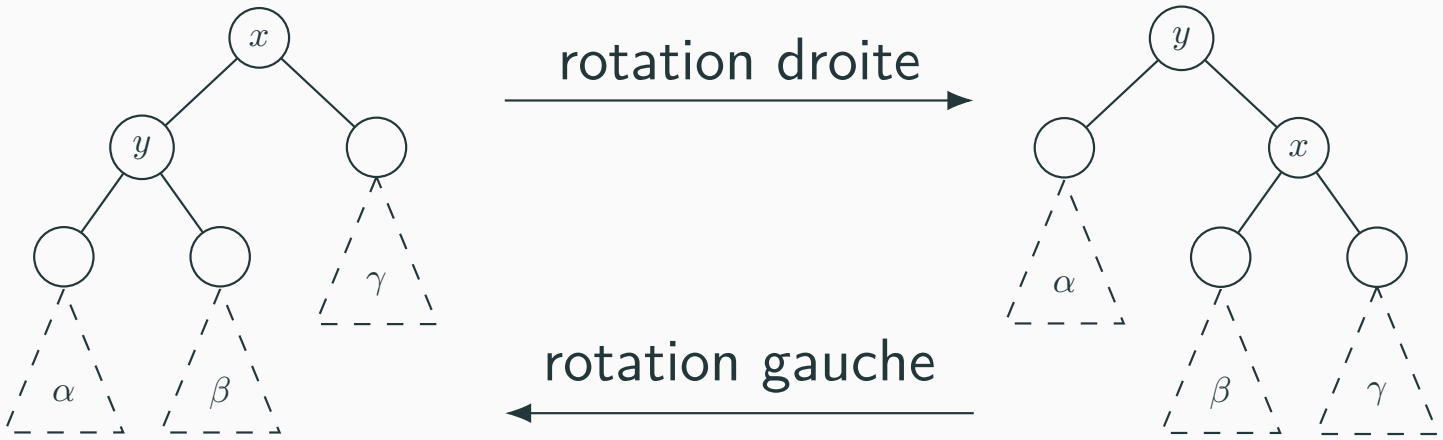
Les opérations de recherche, insertion, et suppression dans un arbre **AVL** à n nœuds se font donc en $O(\log n)$.

Mais il faut modifier les fonctions d'insertion et de suppression pour que la structure d'arbre **AVL** soit maintenue.

On va procéder en utilisant des **rotations** sur l'arbre.

On remarque que ces rotations maintiennent bien la structure d'**ABR**.

Rotations et maintien de la structure d'arbre AVL



Rotations

Rotations dans un **ABR** : x et y sont des nœuds, α, β et γ sont des **ABR**.

Rotations et maintien de la structure d'arbre AVL

Rééquilibrage

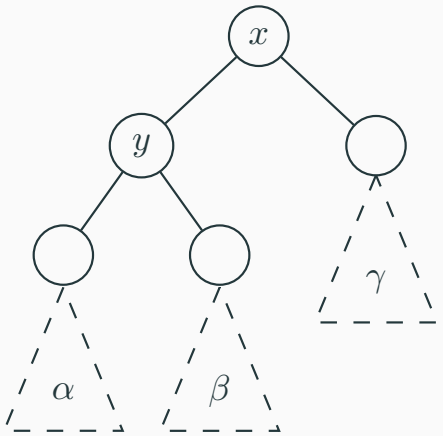
En fait, on va insérer/supprimer les nœuds comme dans les **ABR**, mais on opérera un **rééquilibrage** éventuel "à la remontée", en procédant par des **rotations**.

Lorsqu'on insère/supprime un nœud, il se peut que l'on introduise un déséquilibre qui fait perdre la structure d'**AVL**.

Comme insertion et suppression font varier les hauteurs des sous-arbres d'au plus 1, c'est qu'un nœud (dont le sous-arbre associé a pour hauteur h) possède deux sous-arbres de hauteurs respectives $h-1$ et $h-3$.

On suppose que le sous-arbre de hauteur $h-1$ est celui de gauche, l'autre cas s'en déduit par symétrie.

Rotations et maintien de la structure d'arbre AVL

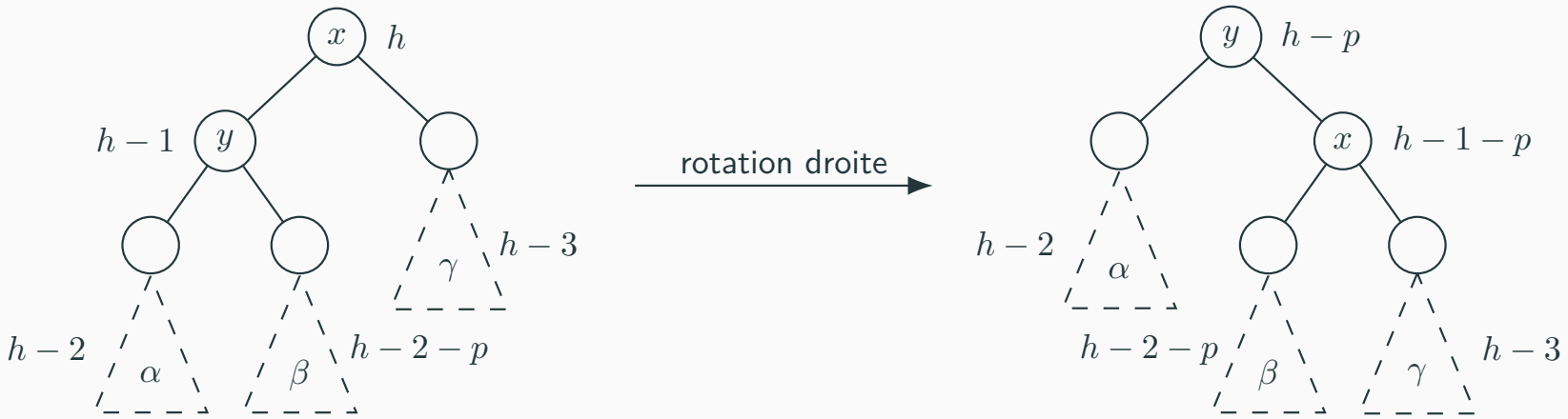


Rééquilibrage

Le sous-arbre de gauche se décompose en une racine y et deux sous-arbres gauche et droit α et β.

On distingue alors deux cas.

Rotations et maintien de la structure d'arbre AVL



Rééquilibrage

Cas 1 : $h(\alpha) \geq h(\beta)$, i.e. $h(\alpha) = h - 2$ et $h(\beta) = h - 2 - p$ avec $p \in \{0, 1\}$.

Dans ce cas, une **rotation droite** suffit, et l'arbre total passe d'une hauteur h à $h - p$: si $p = 1$, le déséquilibre se propage potentiellement plus haut.

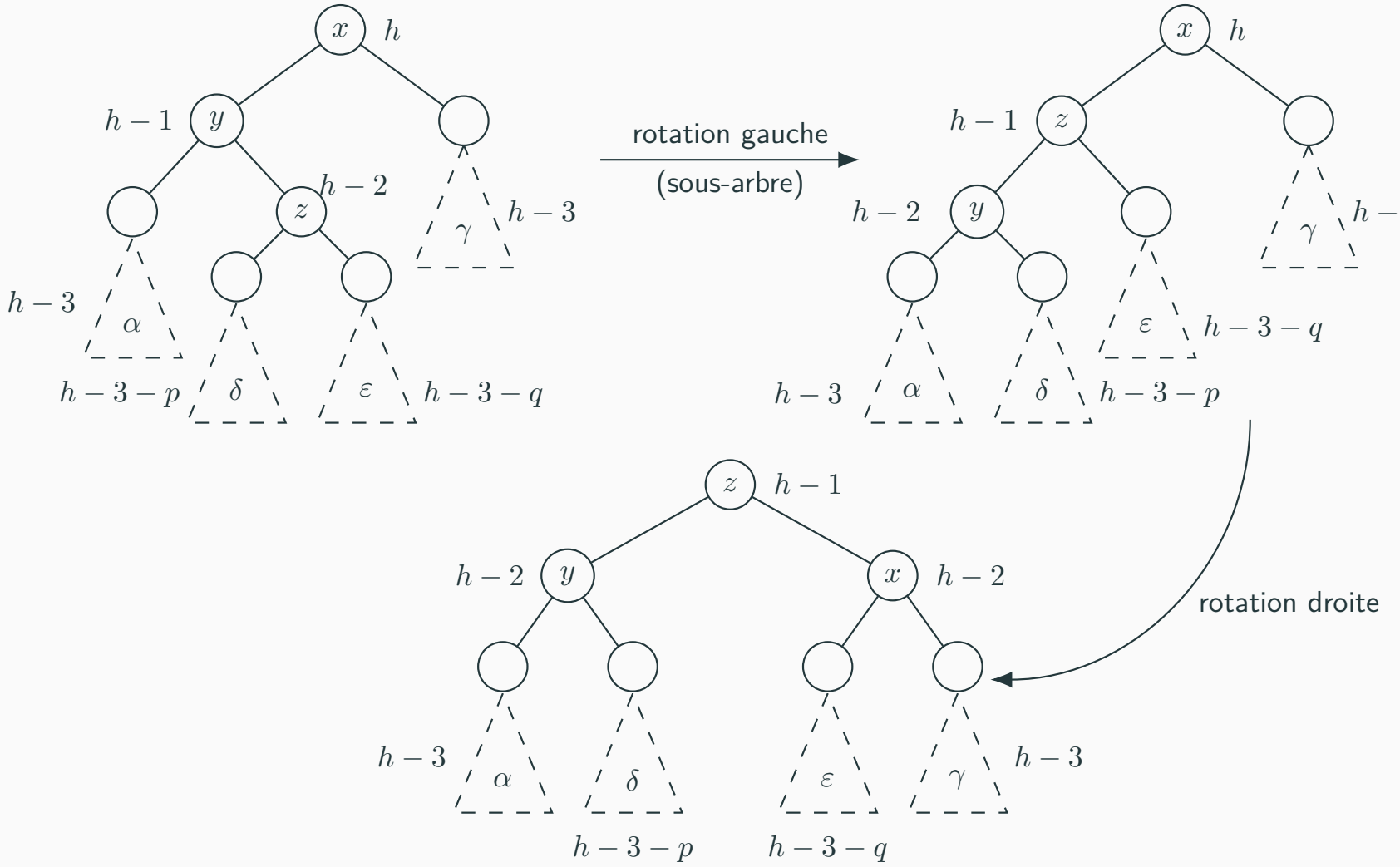
Rééquilibrage

Cas 2 : $h(\alpha) < h(\beta)$. β se décompose en une racine z et deux sous-arbres δ et ε, de hauteur respectives $h-3-p$ et $h-3-q$, avec $p, q \in \{0, 1\}$, l'un au moins étant nul.

On vérifie qu'une **rotation gauche** sur le sous-arbre enraciné en y nous ramène (presque) au cas précédent.

Une **rotation droite** rééquilibre l'arbre total, qui passe d'une hauteur h à une hauteur $h-1$.

Rotations et maintien de la structure d'arbre AVL



Opérations sur les AVL en OCaml

```
1  type 'a avl = Vide | N of int * 'a avl * 'a * 'a avl
```

Idée

On donne simplement l'idée de l'implémentation des opérations sur les **AVL**, qui fera l'objet d'un TP.

Le nouveau champ de type **`int`** indique la **hauteur** du sous-arbre enraciné au nœud.

Ainsi, on n'a pas à recalculer la hauteur des sous-arbres pour savoir s'il faut rééquilibrer l'arbre ou si on est dans le cas 1 ou le cas 2 (cette opération couterait $O(n)$, on perdrait donc la complexité logarithmique des opérations sur les **ABR**).

Opérations sur les AVL en OCaml

Idée

Les schémas précédents permettent d'écrire une fonction `equilibrer : 'a avl -> 'a avl`, qui prend en paramètre un arbre qui est presque un **AVL** : les sous-arbres gauche et droit sont supposés être des **AVL** de hauteur qui diffère d'au plus 2.

Le champ **hauteur** de l'arbre est également possiblement erroné.

La fonction procède si nécessaire à une ou deux rotations, et renvoie un **AVL** (avec champ **hauteur** correct), et ce en **temps constant**.

Opérations sur les AVL en OCaml

Utilisation de la fonction `equilibrer`

Une fois la fonction `equilibrer` écrite, il est facile de réécrire des versions pour arbres **AVL** des fonctions sur les **ABR**.

```
1  let rec inserer a x = match a with
2    | Vide -> N(0,Vide,x,Vide)
3    | N(h,g,y,d) when y=x -> a
4    | N(h,g,y,d) when y<x -> equilibrer (N(h,g,y,inserer d x))
5    | N(h,g,y,d) -> equilibrer (N(h,inserer g x,y,d))
6  ;;
```

Exemple

Voici ci-dessus le code de la fonction d'**insertion**.

Conclusion sur les dictionnaires

Complexité

À l'aide du théorème sur la hauteur d'un **AVL**, on voit que l'on a bien créé une structure de **dictionnaire** où toutes les opérations s'effectuent en $O(\log n)$, où n est le nombre d'éléments stockés.

Tables de hachage

Une autre implémentation classique des **dictionnaires** est celle faisant usage d'une **table de hachage** (cf. DM).