Structure d'arbre

Table des matières

1 Induction informelle 1
- 1.1 Construction des entiers naturels . . . 1
- 1.2 Construction des listes en CAML . . . 1

2 Arbres binaires 2
- 2.1 Définition . . . 2
- 2.2 Arbres binaires étiquetés . . . 2
- 2.3 En CAML . . . 3
- 2.4 Vocabulaire . . . 3
- 2.5 Arbres binaires stricts . . . 3
- 2.6 Arbres binaires étiquetés . . . 4
- 2.7 Relation feuilles - nœuds . . . 4
- 2.8 Indexation des nœuds . . . 5
- 2.9 Profondeur d'un nœud . . . 5
- 2.10 Hauteur d'un arbre . . . 5
- 2.11 Relation entre taille et hauteur . . . 6

3 Parcours d'arbres 7
- 3.1 Définition . . . 7
- 3.2 Parcours en largeur . . . 7
- 3.3 Parcours en profondeur . . . 7

4 Arbres n-aires 8

1 Induction informelle

1.1 Construction des entiers naturels

Définition 1.1

Définition *inductive* :
- 0 est un entier naturel
- Si n est un entier naturel, alors $succ(n)$ aussi

Proposition 1.1

Preuve par récurrence :
Si une propriété $\mathcal{P}_n$ vérifie :
- $\mathcal{P}_0$ est vraie
- $\mathcal{P}_n$ implique $\mathcal{P}_{n+1}$ pour tout n

Alors $\mathcal{P}_n$ est vraie pour tout entier n.

- Calqué sur la définition inductive/récursive.

1.2 Construction des listes en CAML

Définition 1.2

Définition *inductive* :
- `[]` est une liste
- Si `queue` est une liste et `tete` un élément, alors `tete :: queue` est une liste.

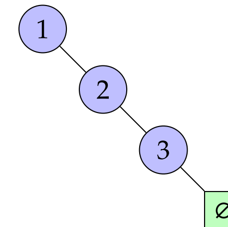


FIGURE 1 – Une représentation possible de la liste `[1; 2; 3]`.

Proposition 1.2

Preuve par induction structurelle :

Soit une propriété $\mathcal{P}_\ell$ sur les listes et une propriété $\mathcal{Q}$ sur les éléments qui vérifient :

- $\mathcal{P}_{[]}$ est vraie
- $\mathcal{P}_{queue} \Rightarrow \mathcal{P}_{tete :: queue}$ pour toute liste `queue` et tout élément `tete` qui vérifie $\mathcal{Q}$.

Alors $\mathcal{P}_\ell$ est vraie pour toute liste ℓ d'éléments qui vérifient $\mathcal{Q}$.

EXERCICE 1

Montrer que la somme des éléments d'une liste d'éléments positifs est positive.

- Récurrence sur la longueur de la liste.
- On dit aussi *récurrence structurelle*.

2 Arbres binaires

2.1 Définition

On peut voir les arbres comme une généralisation de la structure de liste. À une tête (que l'on appelle plutôt **nœud**) on associe non pas une queue, mais deux (que l'on appelle sous-arbres) :

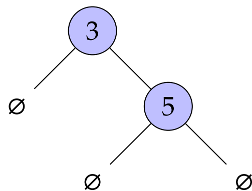


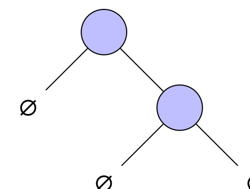
FIGURE 2 – Un exemple d'arbre avec deux nœuds. La racine d'étiquette 3 possède deux sous-arbres : son fils gauche qui est un sous-arbre vide et son fils droit qui est un sous-arbre de taille 1 (une feuille), dont l'étiquette est 5.

Définition 2.1

Définition *inductive* (ou *récursive*) :

- L'arbre vide, noté $\varnothing$ ou `Vide`, est un arbre.
- Si `fils_gauche` et `fils_droit` sont deux arbres, alors un nœud `(fils_gauche, fils_droit)` est un arbre.

Exemple : $(\varnothing, (\varnothing, \varnothing))$



- On appelle **racine** le nœud initial de l'arbre.

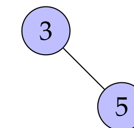
2.2 Arbres binaires étiquetés

Définition 2.2

Définition *inductive* (ou *récursive*) :

- L'arbre vide, noté $\varnothing$ ou `Vide`, est un arbre.
- Si `fils_gauche` et `fils_droit` sont deux arbres et `etiquette` une étiquette, alors un nœud `(etiquette, fils_gauche, fils_droit)` est un arbre.

Exemple : $(3, \varnothing, (5, \varnothing, \varnothing))$



- On ne dessine plus les fils vides.
- Mais attention à l'orientation !

2.3 En CAML

- En CAML on peut définir les arbres binaires avec un type somme :

OCAML

```
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre * 'a arbre
```

OCAML

```
let exemple_arbre =
  Noeud (3, Vide, Noeud (5, Vide, Vide))
```

- Attention : bien respecter la syntaxe de CAML concernant les minuscules et majuscules !

2.4 Vocabulaire

- Un arbre non vide est de la forme

`Noeud(etiquette, fils_gauche, fils_droit).`

- Le nœud est appelé **racine** de l'arbre.
- Les sous-arbres `fils_gauche` et `fils_droits` sont respectivement le **fils gauche** et le **fils droit** de l'arbre.
- Un **fils** d'un nœud est un de ses sous-arbres (suivant le contexte : non vide). Le nœud est alors son **père**.
- Les fils (non vides) d'un même nœud sont appelés **frères**.

Définition 2.3

- On appelle **feuille** (ou nœud externe) un nœud dont les deux sous-arbres sont des arbres vides.
- Un nœud **interne** est un nœud qui n'est pas une feuille.

Définition 2.4

- La **taille** d'un arbre est son nombre de nœuds (internes et externes).
- Le **degré** d'un nœud est son nombre de fils non vides.
- Le **degré** d'un arbre est le maximum des degrés de ses noeuds.

EXERCICE 2

Quelles sont les valeurs possibles pour le degré d'un arbre binaire ? Donner des exemples.

EXERCICE 3

Combien de nœuds n'ont pas de père ? Comment s'appellent-ils ?

2.5 Arbres binaires stricts

Définition 2.5

Un arbre binaire **strict** est un arbre binaire dont tous les nœuds internes sont de degré exactement 2.

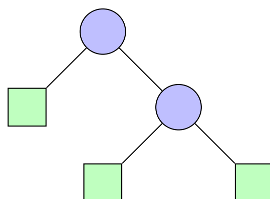
- On trouve parfois la dénomination *complet*, mais celle-ci a aussi et surtout un autre sens !
- Mais le nom *strict* est aussi fait maison.

Définition 2.6

Un arbre binaire *strict* est défini récursivement comme étant

- soit une feuille F,
- soit un nœud, c-à-d un couple (g, d) où g et d sont des arbres binaires stricts

Exemple : $(F,(F,F))$



- On ne peut plus avoir d'arbre vide ! Mais on peut l'ajouter en plus.
- Les arbres binaires seront stricts par la suite.

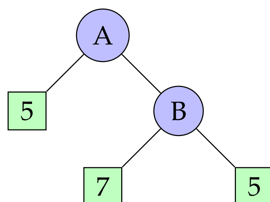
2.6 Arbres binaires étiquetés

On peut choisir de différencier l'information des feuilles et des nœuds.

Définition 2.7

Soit $\mathcal{F}$ un ensemble de valeurs de feuilles et $\mathcal{N}$ un ensemble de valeurs de nœuds. Les arbres binaires (stricts) étiquetés sont définis par récurrence structurelle comme suit :

- Toute feuille, c-à-d tout élément $f \in \mathcal{F}$, est un arbre.
- Si g et d sont des arbres et $n \in \mathcal{N}$ alors le triplet (n,g,d) est un arbre.



- En CAML, le polymorphisme (types paramétrés par d'autres types) permet de définir les arbres étiquetés ainsi

OCAML

```
type ('f, 'n) arbre =
     | Feuille of 'f
     | Noeud of 'n * ('f, 'n) arbre * ('f, 'n) arbre
```

OCAML

```
let exemple_arbre2 =
    Noeud ("A", Feuille 5,
            Noeud ("B", Feuille 7, Feuille 5)
          )
exemple_arbre2 : (int, string) arbre = ...
```

2.7 Relation feuilles - nœuds

Proposition 2.1

Les nombres n_i de nœuds *internes* et f de feuilles d'un arbre binaire **strict** vérifient la relation

$$f = n_i + 1$$

Démonstration

On dénombre les nœuds parents pour chaque nœud interne sauf la racine et chaque feuille : il y en a $f + n_i - 1$.

Mais comme l'arbre est binaire de degré 2, chaque nœud interne est compté exactement deux fois de cette façon. On a donc $f + n_i - 1 = 2n_i$. □

EXERCICE 4

Et pour un arbre binaire non strict ?

Proposition 2.2

Preuve par induction structurelle :

Soit $P(\cdot)$ un prédicat (*i.e.* une propriété) sur les arbres. Supposons

1. P est vrai sur les feuilles : $P(F)$ vrai
2. $P((g,d))$ est vrai dès que $P(g)$ et $P(d)$ sont vrais

alors le prédicat est vrai sur tous les arbres binaires.

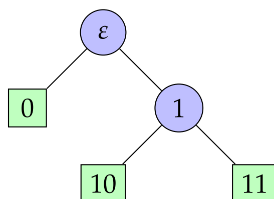
Démonstration : alternative de la relation feuilles nœuds internes $f = n_i + 1$

1. Si l'arbre est une feuille, il y a 0 nœud interne et on a bien $1 = 0 + 1$
2. Si l'arbre est un nœud (g, d), on note n_g, n_d, f_g, f_d les nombres de nœuds internes et feuilles des fils gauche et droit et on suppose que la relation est vérifiée pour les deux fils $f_g = n_g + 1$ et $f_d = n_d + 1$, alors $f = f_g + f_d = n_g + n_d + 1 + 1 = n_i + 1$.

□

2.8 Indexation des nœuds

- Un *mot binaire* est une suite finie de symbole 0 ou 1. Le mot vide est noté ϵ. Exemple : 0010 est un mot binaire de longueur 4.
- Par induction structurelle, on peut associer à chaque nœud d'un arbre un mot binaire de manière injective :
  1. Si l'arbre est une feuille, on lui associe le mot vide ϵ
  2. Si l'arbre est un nœud, on indexe récursivement les fils gauche et droit puis on ajoute comme préfixe le symbole 0 aux index du fils gauche et 1 aux index du fils droit. On associe ϵ à la racine.



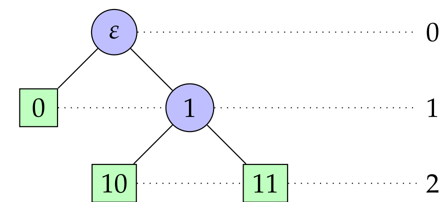
Intuitivement : 0 signifie descendre à gauche et 1 descendre à droite

2.9 Profondeur d'un nœud

Définition 2.8

- La **profondeur** d'un nœud est la longueur du mot binaire qui lui est associé.

Intuitivement, la profondeur d'un nœud est le nombre d'arêtes qu'il faut descendre pour l'atteindre depuis la racine.



Définition 2.9

- Le **niveau de profondeur** $k \geq 0$ d'un arbre est l'ensemble des nœuds de profondeur k.

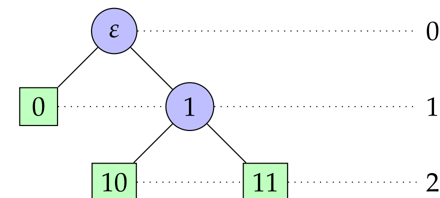
2.10 Hauteur d'un arbre

Définition 2.10

- La **hauteur** d'un arbre est le maximum des profondeurs de ses feuilles (et nœuds).
- La **hauteur** d'un nœud est la hauteur du sous-arbre enraciné en ce nœud.

La hauteur d'un arbre est donc la hauteur de sa racine.

Intuitivement, la hauteur d'un nœud est le nombre maximal d'arêtes que l'on peut descendre pour atteindre la feuille la plus profonde (de ce sous-arbre).



Proposition 2.3

La fonction hauteur h vérifie les relations

1. $h(F) = 0$ dans le cas de l'arbre feuille
2. $h((g,d)) = 1 + \max(h(g), h(d))$ sinon.

Démonstration

Par induction structurelle. □

- en CAML, on écrit grâce au filtrage de motif et à la récursivité :

OCAML

```
let rec hauteur arbre =
  match arbre with
  | Feuille -> 0
  | Noeud (fg, fd) -> 1 + max (hauteur fg) (hauteur fd)
```

☞ **Remarque 1 (*importante*)**

La définition inductive des arbres binaires impose que ceux-ci ont **toujours** *un nombre de nœuds et de feuilles* **fini**. *En particulier*

- *un arbre binaire a toujours une* **taille finie**,
- *un arbre binaire a toujours une* **hauteur finie**.

- Ce cours ne traite donc pas des arbres infinis.

2.11 Relation entre taille et hauteur

Proposition 2.4

Soit n_i le nombre de nœuds internes d'un arbre binaire strict et h sa hauteur alors

$$\log_2(n_i + 1) \leq h \leq n_i$$

Démonstration

- Un chemin menant de la racine à une feuille de profondeur maximale contient h nœuds internes donc

$$h \leq n_i$$

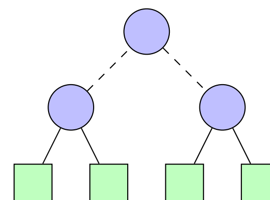
- Il existe 2^p mots binaires de longueur p. D'après notre indexation injective par les mots binaires, un arbre de profondeur h ne peut donc compter au maximum que (l'index des nœuds internes étant de longueur maximale $h - 1$)

$$\sum_{p=0}^{h-1} 2^p = 2^h - 1 \text{ nœuds internes.}$$

Donc $n_i \leq 2^h - 1$ d'où $\log_2(n_i + 1) \leq h$.

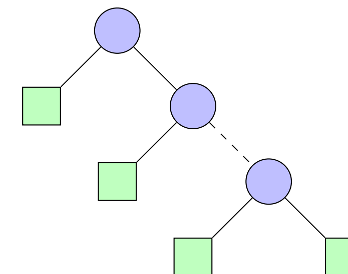
(Se voit aussi en comptant le nombre de nœuds maximum sur chaque niveau.) □

Arbre complet



$h = \log_2(n_i + 1)$

Arbre peigne



$h = n_i$

☞ **Remarque 2 (*Conséquence*)**

Nous verrons que la complexité des algorithmes sur les arbres s'exprime souvent en fonction de leur hauteur. Les algorithmes seront donc plus efficaces sur les arbres équilibrés ($h \approx \log_2 n_i$) que sur les arbres déséquilibrés ($h \approx n_i$).

3 Parcours d'arbres

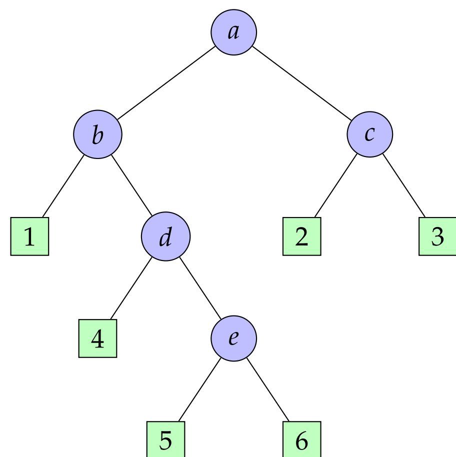
3.1 Définition

Définition 3.1

Un parcours d'arbres consiste à visiter/imprimer/lister/utiliser tous ses nœuds et feuilles dans un ordre prédéterminé.

- Les parcours d'arbres sont adaptés lorsqu'on souhaite appliquer un traitement à tous les nœuds et/ou feuilles d'un arbre :
  - afficher toutes les valeurs de l'arbre
  - vérifier la présence d'une valeur
  - chercher la plus grande valeur
  - ...
- TD : peut-on reconstruire l'arbre à partir de son parcours (parcours non ambigu) ?

3.2 Parcours en largeur



a b c 1 d 2 3 4 e 5 6

Définition 3.2

Le **parcours en largeur** consiste à visiter les éléments
- par profondeur croissante
- de gauche à droite

- Ce parcours simple est toutefois assez éloigné de la structure récursive...
- Pour plus tard !

3.3 Parcours en profondeur

Les parcours en profondeur exploitent la définition récursive des arbres. Il en existe trois principaux :

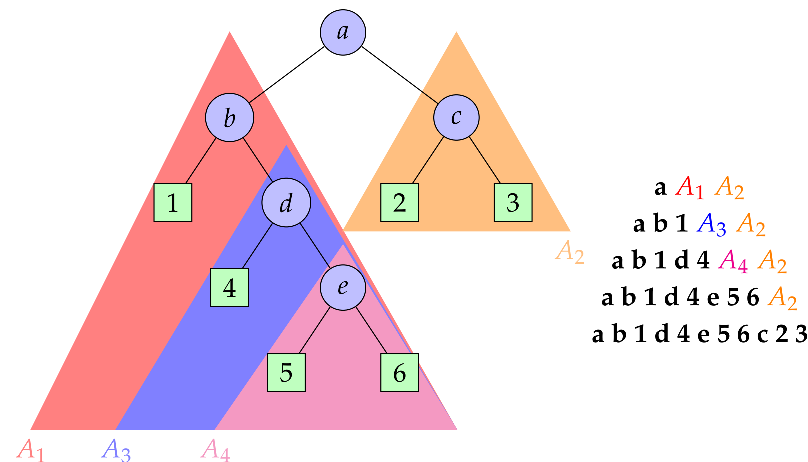
Définition 3.3

Parcours préfixe
1. Traiter la racine
2. Parcourir récursivement le fils gauche
3. Parcourir récursivement le fils droit

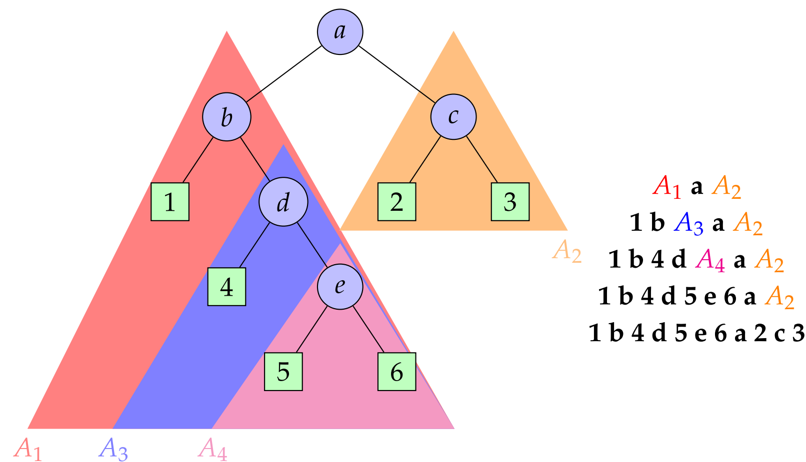
Parcours infixe
1. Parcourir récursivement le fils gauche
2. Traiter la racine
3. Parcourir récursivement le fils droit

Parcours suffixe
1. Parcourir récursivement le fils gauche
2. Parcourir récursivement le fils droit
3. Traiter la racine

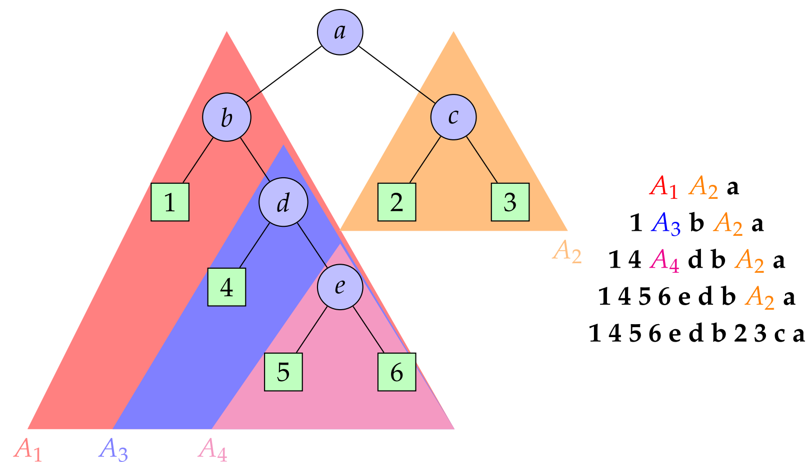
- Le parcours d'une feuille est évident.
- Les parcours en profondeur se distinguent donc essentiellement par le choix du moment où l'on traite la racine (d'un sous-arbre).
- **Parcours préfixe** :



- **Parcours infixe** :



- **Parcours suffixe (postfixe)** :



- Les parcours en profondeur étant définis selon la structure récursive des arbres, ils sont faciles à programmer en CAML.

- Parcours préfixe :

OCAML
```
let rec parcours_prefixe arbre =
  match arbre with
   | Feuille etiquette -> print_int etiquette
   | Noeud (etiquette, fils_gauche, fils_droit) ->
     print_int etiquette;
     parcours_prefixe fils_gauche;
     parcours_prefixe fils_droit
```

- Que faut-il changer pour avoir un parcours infixe/postfixe ?
- Et si on souhaite une fonction de type `'a arbre -> 'a list` qui renvoie la liste au lieu d'imprimer les éléments ?

OCAML
```
let rec parcours_prefixe arbre =
  match arbre with
   | Feuille etiquette -> [etiquette]
   | Noeud (etiquette, fils_gauche, fils_droit) ->
     [etiquette] @ (parcours_prefixe fils_gauche)
                 @ (parcours_prefixe fils_droit)
```

- Que faut-il changer pour avoir un parcours infixe/postfixe ?

⚠ L'utilisation de la concaténation de listes `@` n'est pas efficace. Nous y reviendrons !

4 Arbres n-aires

Les arbres n-aires généralisent les arbres binaires en permettant d'avoir autant de fils souhaités à chaque nœud.

Définition 4.1

Un arbre n-aire sur un ensemble de valeurs de nœuds $\mathcal{X}$ est un couple $(x, (a_1, \ldots, a_k))$ avec

- $x \in \mathcal{X}$ l'étiquette du nœud (racine)
- $k \in \mathbb{N}$ l'*arité* du nœud
- $a_1, \ldots, a_k$ des arbres n-aires

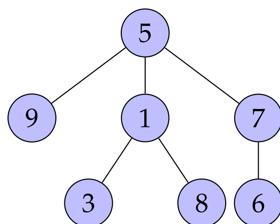
☞ Remarque 3

- *Le n n'a pas de sens.*
- *L'équivalent d'une feuille est donc un nœud d'arité 0.*

En CAML on peut définir les arbres n-aires ainsi :

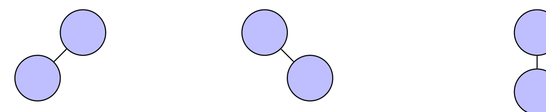
OCAML
```
type 'a arbre = Noeud of 'a * ('a arbre list)

let exemple =
  Noeud (5, [Noeud (9, []);
             Noeud (1, [Noeud (3, []); Noeud (8, [])]);
             Noeud (7, [Noeud (6, [])])
            ])
```



Quand on considère les arbres n-aires, les notions de **fils gauche et droit n'existent plus**. Seule la notion de fils numéro i existe.

En particulier, si on considère la définition d'arbres binaires sans feuille, on ne peut pas distinguer les deux premiers arbres ci-dessous, car leur équivalent en arbre n-aire est le troisième.



Les notions de taille et profondeur existent toujours sur les arbres n-aires. On peut les programmer en CAML à l'aide de deux fonctions mutuellement récursives :

OCAML
```
let rec taille (Noeud (_, fils)) =
  1 + taille_foret fils
and taille_foret foret =
  match foret with
  | [] -> 0
  | arbre :: autres ->
      (taille arbre) + (taille_foret autres)
```

☞ Remarque 4

*Une **forêt** est un ensemble d'arbres, ici représentée par une liste d'arbres.*

EXERCICE 5

Écrire la fonction `hauteur : 'a arbre -> int`. On pourra définir en parallèle la fonction `max_hauteur_foret : 'a arbre list -> int`.

De même, les parcours préfixe et suffixe (mais pas infixe) peuvent être redéfinis sur les arbres n-aires. Par exemple :

OCAML
```
let rec parcours_prefixe (Noeud (etiquette, fils)) =
  print_int etiquette;
  parcours_foret fils
and parcours_foret foret =
  match foret with
  | [] -> ()
  | arbre :: autres ->
     parcours_prefixe arbre;
     parcours_foret autres
```