Arbres binaires #1

Pierre Karpman

Lycée Champollion MP2I
https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Arbre binaires : définition

Dessin

Définition *inductive*

- Cas de base : un arbre vide est un arbre sur n'importe quel type α
- Cas inductif : si t_1 et t_2 sont des arbres sur un même type α, $x : \alpha$, l'arbre formé par x et les enfants t_1 et t_2 est un arbre sur α
- On considère le plus petit ensemble ainsi construit

Implémentation comme un type récursif OCaml

```
type 'a tree = E | N of 'a * 'a tree * 'a tree
```

Arbres binaires : vocabulaire de base

Nœuds etc.

- *Nœud* : le produit du constructeur `N`
- *Racine* : le nœud « initial » : un nœud *n* tel qu'il n'existe aucun nœud *n'* avec `n' = N (_, n, _)` (ou `n' = N (_, _, n)`)
- *Feuille* : un nœud de la forme `N (_, E, E)`
- *Nœud interne* : un nœud qui n'est pas une feuille

Enfants, parents…

`let n = N (x, lc, rc)` :

- *x* est l'*étiquette* de *n*
- le *sous-arbre lc* est l'*enfant gauche* (vocabulaire officiel : *fils gauche*) de *n*,
- — *rc* — *droit* (— *droit*) —
- *n* est le *parent* (vocabulaire officiel : *père*) de *lc* & *rc*

Arbres binaires : vocabulaire de base *bis*

Taille, degré

- La *taille* d'un arbre est son nombre de nœuds
- Le *degré* d'un nœud est son nombre d'enfants non vides ; le *degré* d'un arbre est le degré max de ses nœuds
  - À quoi correspond un arbre de degré 1 ?

Taille : implémentation en OCaml ?

Arbre binaires : hauteur, profondeur

Hauteur d'un arbre : définition inductive

```
let rec height
    = function
    | E -> -1
    | N (_, lc, rc) -> 1 + max (height lc) (height rc)
```

Pas de consensus sur la hauteur d'un arbre vide (`-1` : choix du programme, mais…)

Profondeur d'un nœud

La profondeur d'un nœud d'un arbre A est la longueur du chemin qui le relie à la racine.

- Niveau de profondeur p : tous les nœuds de même profondeur

Hauteur d'un arbre : définition par profondeur

La hauteur d'un arbre est la profondeur maximale de ses nœuds.

Arbres binaires : arbres particuliers

Arbre binaire *strict*

Un arbre binaire dont tous les nœuds ont degré 0 ou 2 (ou : dont tous les nœuds internes ont deux enfants)

Implémentation « forcée » comme un type récursif OCaml

```
type ('a, 'b) tree2 =
| L of 'a (* leaf *)
| IN of 'b * ('a, 'b) tree2 * ('a, 'b) tree2 (* internal node *)
```

- Zéro ou deux enfants

Arbre binaire *complet*

Arbre binaire strict dont toutes les feuilles sont à la même profondeur

- Pas nécessairement représenté par le type `('a, 'b) tree2`
- Parfois aussi appelé *arbre parfait*

Quelques propriétés sur les arbres binaires

Relation nœuds-feuille

Soit N le nombre de nœuds internes d'un arbre binaire A, F son nombre de feuilles, alors $F \leq N + 1$, avec égalité ssi. A est strict

Nombre de nœuds d'un arbre complet

Le nombre (total) de nœuds d'un arbre binaire complet de hauteur h est $2^{h+1} - 1$

Preuves

En TD ?

Table des matières

Numérotation/adressage/indexation des nœuds

Il peut être pratique de définir une numérotation des nœuds d'un arbre (binaire)

Approche « mots binaires »

Les *mots binaires* sont définis inductivement comme :

- le mot vide ε, neutre pour la concaténation $\cdot$
- soit m un mot binaire, $0 \cdot m$, $1 \cdot m$, $m \cdot 0$, $m \cdot 1$ sont des mots binaires resp. obtenus par concaténation gauche et droite de la lettre 0 et 1 au mot m

On indexe inductivement les nœuds d'un arbre binaire par des mots binaires :

- La racine a index ε (variante : 1)
- `let n = N (_, lc, rc)` d'index ν_n on définit :
  - $\nu_{\text{lc}} := 0 \cdot \nu_n$ (variante : $\nu_n \cdot 0$)
  - $\nu_{\text{rc}} := 1 \cdot \nu_n$ (variante : $\nu_n \cdot 1$)

Intuitivement : un index donne le chemin à suivre depuis la racine (0 : prendre à gauche, 1 : prendre à droite)

Numérotation « mots binaires » : applications

- La profondeur d'un nœud est la longueur (en tant que mot binaire) de son index
- Le niveau de profondeur p est l'ensemble des nœuds d'index de longueur p
- La hauteur d'un arbre est la longueur maximale des index de ses descendants
  - Re : un arbre binaire complet de hauteur h possède $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ nœuds

Relation taille/hauteur

Soit n_i, h respectivement le nombre de nœuds internes et la hauteur d'un arbre binaire, on a :

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

Preuve :

- Un arbre de hauteur h possède une feuille d'index de longueur h, qui implique l'existence de h nœuds internes depuis la racine
- Le nombre de nœuds interne est $\leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$

Numérotation « mots binaires » : applications *bis*

Implémentation de tableaux fonctionnels plutôt efficaces (*cf.* un TP)

Vocabulaire, remarques

Vocabulaire

- Arbre peigne : $h = n_i$
- Arbre équilibré (informel) : $h \approx \log(n_i)$
- Arbre déséquilibré (informel) : $h \approx n_i$

Arbre binaire v. liste

- « Ce que la base deux est à la base un » pour le stockage d'information

Coût

Beaucoup d'algorithmes d'arbres (mais pas tous) sont de coût « égal » (asymptotiquement, à constantes près) à la *hauteur* de l'arbre

- À même nombre de nœuds, exponentiellement plus efficaces sur un arbre équilibré que sur l'arbre peigne
- En pratique, il y a « beaucoup plus » d'arbres équilibrés que d'arbres déséquilibrés
  - les choses se passent bien *en moyenne* (mais qu'est-ce que cela veut dire ?)
  - aussi possible de garantir (évt. probabilistiquement) que les arbres manipulés sont toujours équilibrés (*cf.* plus tard)

Numérotation/adressage/indexation des nœuds

Approche naturelle

On numérote inductivement les nœuds d'un arbre binaire par des entiers naturels :

- La racine a numéro 0
- `let n = N (_, lc, rc)` de numéro ν_n on définit :
  - $\nu_{\text{lc}} := 2 \times \nu_n + 1$
  - $\nu_{\text{rc}} := 2 \times \nu_n + 2$

Cette numérotation permet notamment de stocker facilement les arbres dans des tableaux !

- Un exemple primitif de *sérialisation*, très adapté à certains contextes
- *Cf.* bientôt

Parcours d'arbres

Un parcours d'arbre est un algorithme visitant les nœuds d'un arbre afin de leur appliquer un traitement

Contraintes :

- (Pouvoir) visiter *tous* les nœuds
  - une unique fois
  - éventuellement dans un ordre précis

Parcours en profondeur

Récursivement ; basé sur la définition inductive des arbres

- Trois variantes (*cf. infra*)

Parcours en largeur

Parcours les nœuds dans l'ordre des niveaux de profondeur

- Typiquement implémenté avec une file

Parcours en profondeur (gauche)

Trois variantes en fonction de l'ordre de traitement de la racine

Parcours préfixe (gauche)

1. Traiter la racine
2. Traiter récursivement le sous-arbre (gauche)
3. Traiter récursivement le sous-arbre (droit)

Parcours infixe (gauche)

1. Traiter récursivement le sous-arbre (gauche)
2. Traiter la racine
3. Traiter récursivement le sous-arbre (droit)

Parcours postfixe (ou suffixe) (gauche)

1. Traiter récursivement le sous-arbre (gauche)
2. Traiter récursivement le sous-arbre (droit)
3. Traiter la racine

Parcours en profondeur : implémentation

Exemple :

```
let rec iter_dfsa (f : 'a -> unit)
    = function
    | E -> ()
    | N (x, lc, rc) -> f x ; iter_dfsa f lc ; iter_dfsa f rc
```

Parcours en profondeur : Remarques

Rappel

Lors d'appels récursifs multiples dans une fonction récursive, représentés par un arbre

- l'ordre d'appel (l'empilement) est donné par un parcours en profondeur préfixe
- l'ordre de retour (le dépilement) est donné par un parcours en profondeur postfixe

Futurs exemples d'application

- Arbre binaire de recherche : énumérer les nœuds dans l'ordre : parcours infixe
- Arbre d'une expression arithmétique : évaluation : parcours postfixe

Table des matières

Preuves sur les arbres

Les preuves sur les arbres se font souvent par

- Récurrence (le plus simple à rédiger ?)
  - Le plus souvent par récurrence forte sur la hauteur
- *Induction structurelle* (le plus algorithmique ?)
  - On montre que la propriété est vraie pour les constructeurs de base, et préservée par les constructeurs inductifs
  - Preuve $\approx$ filtrage de motif exhaustif
  - Plus de détails un peu plus tard…

Table des matières

Représentation en C

Par exemple *via* un type :

```c
struct btree
{
    struct btree *parent; // optionnel
    struct btree *left;
    struct btree *right;
    int data; // ou void* pour être générique
};
```

- Simple généralisation arborescente d'une représentation de liste simplement / doublement chaînée
- L'absence d'un parent ou d'un enfant `left` / `right` s'implémente en utilisant une valeur nulle pour le pointeur concerné
- On peut représenter de tels arbres mutables avec des enregistrement OCaml, sur le même mode