

## Induction sur les arbres

### Planning

- Motivations
- Comment définir les arbres ?
- Équations récursives sur les arbres
- Complexité de fonctions sur les arbres
  - Recherche dans un arbre binaire de recherche
  - Recherche dans un arbre 2-3-4
- Preuve de propriétés par récurrence sur les arbres

### Les arbres en informatique

- Organiser des données en une structure hiérarchique (les fichiers dans un système d'exploitation, le sommaire d'un livre, les expressions arithmétiques, les phrases du langage naturelle, les arbres de jeu)
- Rechercher/accéder à l'information plus facilement
- Modéliser de nombreux problèmes

### Quelques types d'arbres

Arbre binaire (I) : tout nœud interne a **exactement** deux fils.

Arbre binaire (II) : tout nœud interne a **au plus** deux fils.

Arbre n-aire (I) : tout nœud interne a **exactement**  $n$  fils.

Arbre n-aire (II) : tout nœud interne a **au plus**  $n$  fils.

Arbre quelconque : le nombre de fils **n'est pas borné**.

### Les arbres binaires de type I sans étiquette



L'ensemble *AbinSE* est le plus petit ensemble t.q.

- *avide* ∈ *AbinSE*
- Si  $a_1, a_2$  ∈ *AbinSE*, alors  $noeud(a_1, a_2)$  ∈ *AbinSE*.

1

```
# arbre_gauche arbl ;;
- : abinse = N (Av, Av)
```

```
# arbre_droite arbl ;;
- : abinse = Av
```

### Équations récursives pour AbinSE

Schéma récursif :

$$f(a) = g(a, f(\text{fils\_gauche}(a)), f(\text{fils\_droit}(a)))$$

### Exemple : nombre de feuilles

**Problème** : définir une fonction qui calcule le nombre de feuilles d'un arbre binaire de type I.

**Déclaration du type** :

```
nb_f : AbinSE → entier
```

**Équation récursive** :

$$nb\_f(a) = g(a, nb\_f(\text{fils\_gauche}(a)), nb\_f(\text{fils\_droit}(a)))$$

**Cas particulier** :

a=avide

On pose

$$g(a, z_1, z_2) = z_1 + z_2$$

Ceci donne :

$$nb\_f(a) = \begin{cases} 1 & \text{si } est\_arbre\_vide(a) \\ nb\_f(\text{fils\_gauche}(a)) + nb\_f(\text{fils\_droit}(a)) & \text{sinon} \end{cases}$$

### Le nombre de feuilles en OCAML

```
# let rec nb_feuilles a = match a with
  Av      -> 1
  | N(x,y) -> nb_feuilles x + nb_feuilles y;;
val nb_feuilles : abinse -> int = <fun>
```

3

### Un type abstrait AbinSE

Domaine :

AbinSE

Opérations de construction :

avide : AbinSE

noeud : AbinSE × AbinSE → AbinSE

Opérations de test :

est\_arbre\_vide : AbinSE → booléen

Opérations d'accès :

fils\_gauche : AbinSE → AbinSE

fils\_droit : AbinSE → AbinSE

### AbinSE en OCAML

```
# type abinse = Av | N of abinse * abinse;;
type abinse = Av | N of abinse * abinse

(* opération de construction *)
# let cons_ab(x,y) = N(x,y);;
val cons_ab : abinse * abinse -> abinse = <fun>

(* opérations d'accès *)
# let fils_gauche a = match a with
  N(x,y) -> x
  | _      -> failwith "erreur";;
val fils_gauche : abinse -> abinse = <fun>

# let fils_droite a = match a with
  N(x,y) -> y
  | _      -> failwith "erreur";;
val fils_droite : abinse -> abinse = <fun>

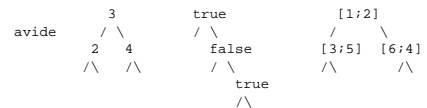
(* opération de test *)
# let est_arbre_vide a = match a with
  Av -> true | _ -> false;;
val est_arbre_vide : abinse -> bool = <fun>

# let arbl = N(N(Av,Av),Av);;
val arbl : abinse = N (N (Av, Av), Av)

# est_arbre_vide arbl;;
- : bool = false
```

2

### Les arbres binaires de type I avec étiquette de type 'a



L'ensemble 'a abin est le plus petit ensemble t.q.

- *avide* ∈ 'a abin
- Si  $a_1, a_2$  ∈ 'a abin et  $m$  est de type 'a, alors  $abr(m, a_1, a_2)$  ∈ 'a abin.

### Un type abstrait pour 'a abin

Domaine :

'a abin

Opérations de construction :

avide : 'a abin

abr : 'a × 'a abin × 'a abin → 'a abin

Opérations de test :

est\_arbre\_vide : 'a abin → booléen

Opérations d'accès :

etiquette : 'a abin → 'a

fils\_gauche : 'a abin → 'a abin

fils\_droit : 'a abin → 'a abin

### 'a abin en OCAML

```
# type 'a abin = Av | Arb of 'a * 'a abin * 'a abin;;
type 'a abin = Av | Arb of 'a * 'a abin * 'a abin

(* opération de construction *)
# let cons_ab(e,x,y) = Arb(e,x,y);;
val cons_ab : 'a * 'a abin * 'a abin -> 'a abin = <fun>

(* opérations d'accès *)
# let etiquette a = match a with
  Arb(e,x,y) -> e
  | _          -> failwith "erreur";;
val etiquette : 'a abin -> 'a = <fun>
```

4

```

# let fils_gauche a = match a with
  Arb(e,x,y) -> x
  | _ -> failwith "erreur";;
val fils_gauche : 'a abin -> 'a abin = <fun>

# let fils_droite a = match a with
  Arb(e,x,y) -> y
  | _ -> failwith "erreur";;
val fils_droite : 'a abin -> 'a abin = <fun>

(* opération de test *)
# let est_arbre_vider a = match a with
  Av -> true | _ -> false;;
val est_arbre_vider : 'a abin -> bool = <fun>

# let arbl = Arb(4,Arb(2,Av,Av),Av);;
val arbl : int abin = Arb (4, Arb (2, Av, Av), Av)

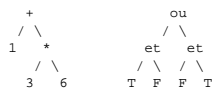
# let arb2 = Arb("bonjour",Arb("madame",Av,Av),Av);;
val arb2 : string abin = Arb ("bonjour", Arb ("madame", Av, Av), Av)

# etiquette arbl;;
- : int = 4

# etiquette arb2;;
- : string = "bonjour"

```

### Les arbres binaires de type II avec étiquette



- On les utilise par exemple pour représenter des expressions arithmétiques.
- Pas de notion d'arbre vide : l'expression "la plus petite" est une étiquette.

5

```

val eti_feuille : ('a, 'b) abinII -> 'a = <fun>

# let eti_noeud a = match a with
  N(e,x,y) -> e
  | _ -> failwith "erreur";;
val eti_noeud : ('a, 'b) abinII -> 'b = <fun>

# let fils_gauche a = match a with
  N(e,x,y) -> x
  | _ -> failwith "erreur";;
val fils_gauche : ('a, 'b) abinII -> ('a, 'b) abinII = <fun>

# let fils_droite a = match a with
  N(e,x,y) -> y
  | _ -> failwith "erreur";;
val fils_droite : ('a, 'b) abinII -> ('a, 'b) abinII = <fun>

(* opération de test *)
# let est_feuille a = match a with
  F(_) -> true | _ -> false;;
val est_feuille : ('a, 'b) abinII -> bool = <fun>

# type op_arith = Plus | Mul | Sous | Div ;;
type op_arith = Plus | Mul | Sous | Div

# let arbl = N(Mul,F(4),F(5));;
val arbl : (int, op_arith) abinII = N (Mul, F 4, F 5)

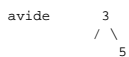
# let arb2 = N(Plus,F(4.4),F(5.3));;
val arb2 : (float, op_arith) abinII = N (Plus, F 4.4, F 5.3)

```

### Les arbres n-aires

On généralise sans problèmes les arbres binaires de type I ou II au cas n-aire.

### Les arbres quelconques



7

- Chaque nœud est unaire ou binaire.

### Les arbres binaires de type II avec étiquette

C'est un ensemble paramétré par deux types 'a et 'b, où chaque étiquette d'une feuille est de type 'a et chaque étiquette d'un nœud interne est de type 'b.

L'ensemble ('a, 'b) AbinII est le plus petit ensemble t.q.

- Si m est de type 'a, alors feuille(m) ∈ ('a, 'b) AbinII
- Si a<sub>1</sub>, a<sub>2</sub> ∈ ('a, 'b) AbinII et o est de type 'b, alors noeud(o, a<sub>1</sub>, a<sub>2</sub>) ∈ ('a, 'b) AbinII.

### Un type abstrait pour ('a, 'b) abinII

**Domaine :**  
('a, 'b) abinII  
**Opérations de construction :**  
cons\_f : 'a → ('a, 'b) abinII  
cons\_n : 'b × ('a, 'b) abinII × ('a, 'b) abinII → ('a, 'b) abinII  
**Opérations de test :**  
est\_feuille : ('a, 'b) abinII → booléen  
**Opérations d'accès :**  
etiquette\_feuille : ('a, 'b) abinII → 'a  
etiquette\_noeud : ('a, 'b) abinII → 'b  
fils\_gauche, fils\_droit : ('a, 'b) abinII → ('a, 'b) abinII  
**('a, 'b) abinII OCAML**

```

# type ('a, 'b) abinII =
  F of 'a | N of 'b * ('a, 'b) abinII * ('a, 'b) abinII;;

```

```

(* opération de construction *)
# let cons_feuille(e) = F(e);;
val cons_feuille : 'a -> ('a, 'b) abinII = <fun>

```

```

# let cons_noeud(c,x,y) = N(c,x,y);;
val cons_noeud :

```

```

'b * ('a, 'b) abinII * ('a, 'b) abinII -> ('a, 'b) abinII = <fun>

```

```

(* opérations d'accès *)
# let eti_feuille a = match a with
  F(e) -> e
  | _ -> failwith "erreur";;

```

6



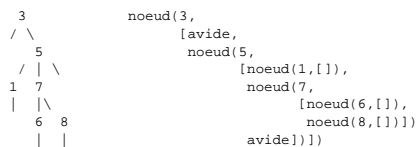
### Les arbres quelconques : définition

- Un arbre quelconque est
- soit un arbre vide
  - soit un nœud interne ayant un nombre arbitraire de fils

L'ensemble 'a arbre est le plus petit ensemble t.q.

- avide ∈ 'a arbre
- Si m ∈ 'a et l<sub>i</sub> ∈ ('a arbre) liste, alors noeud(m, l) ∈ 'a arbre.

### Un exemple



### Type abstrait pour un arbre quelconque

**Domaine :**  
'a arbre  
**Opérations de construction :**  
avide : 'a arbre  
cons\_arbre : 'a × ('a arbre) liste → 'a arbre  
**Opérations de test :**  
est\_arbre\_vider : 'a arbre → booléen  
**Opérations d'accès :**  
racine : 'a arbre → 'a  
fils : 'a arbre → ('a arbre) liste  
**En OCAML**

```

# type 'a arbre = Av | N of 'a * 'a arbre list ;;

```

8

```

type 'a arbre = N of 'a * 'a arbre list

(* opération de construction *)
# let cons_arbre_vider = Av;
val cons_arbre_vider : 'a arbre = Av

# let cons_noeud(e,l) = N(e,l);
val cons_noeud : 'a * 'a arbre list -> 'a arbre = <fun>

(* opération de test *)
# let est_arbre_vider a = match a with
  Av -> true | _ -> false;;
val est_arbre_vider : 'a arbre -> bool = <fun>

(* opérations d'accès *)
# let racine a = match a with
  N(e,l) -> e
  | _ -> failwith "erreur";;
val racine : 'a arbre -> 'a = <fun>

# let fils a = match a with
  N(e,l) -> l
  | _ -> failwith "erreur";;
val fils : 'a arbre -> 'a arbre list = <fun>

```

### Équations récursives sur les arbres quelconques

Schéma récursif :

$$\begin{aligned}
 f(a) &= g_1(a, f_F(\text{fils}(a))) \\
 f_F(s) &= g_2(s, f(\text{premier}(s)), f_F(\text{reste}(s)))
 \end{aligned}$$

Le domaine de  $f$  est  $'a \text{ arbre}$

Le domaine de  $f_F$  est  $('a \text{ arbre}) \text{ liste}$

Cas particuliers :

$a = \text{avide}$  (fils(avide) n'est pas défini).  
 $s = \text{nil}$  (premier(nil) et reste(nil) ne sont pas définis).

### Exemple I : la taille d'un arbre quelconque

**Problème :** définir une fonction qui calcule le nombre de nœuds internes d'un arbre quelconque.

**Déclaration du type :**

9

$$\begin{aligned}
 \text{taille} : 'a \text{ arbre} \rightarrow \text{entier} \\
 \text{taille}_F : ('a \text{ arbre}) \text{ liste} \rightarrow \text{entier}
 \end{aligned}$$

Équation récursive :

$$\begin{aligned}
 \text{taille}(a) &= g_1(a, \text{taille}_F(\text{fils}(a))) \\
 \text{taille}_F(s) &= g_2(s, \text{taille}(\text{premier}(s)), \text{taille}_F(\text{reste}(s)))
 \end{aligned}$$

On pose

$$g_1(a, z) = 1 + z \quad g_2(a, z_1, z_2) = z_1 + z_2$$

Ceci donne :

$$\begin{aligned}
 \text{taille}(a) &= 0 && \text{si est\_arbre\_vider}(a) \\
 \text{taille}(a) &= 1 + \text{taille}_F(\text{fils}(a)) && \text{sinon}
 \end{aligned}$$

$$\begin{aligned}
 \text{taille}_F(s) &= 0 && \text{si liste\_vider}(s) \\
 \text{taille}_F(s) &= \text{taille}(\text{premier}(s)) + \text{taille}_F(\text{reste}(s)) && \text{sinon}
 \end{aligned}$$

### La taille d'un arbre en OCAML

```

# let rec taille a = match a with
  Av -> 0
  | N(_,l) -> 1 + taille_f(l)
and
taille_f l = match l with
  [] -> 0
  | p::r -> taille p + taille_f r;;
val taille : 'a arbre -> int = <fun>
val taille_f : 'a arbre list -> int = <fun>

```

### Exemple II : les étiquettes d'un arbre quelconque

**Problème :** définir une fonction qui calcule l'ensemble des valeurs associées aux nœuds d'un arbre quelconque.

**Déclaration du type :**

10

$$\begin{aligned}
 \text{ensval} : 'a \text{ arbre} \rightarrow 'a \text{ ens} \\
 \text{ensval}_F : ('a \text{ arbre}) \text{ liste} \rightarrow 'a \text{ ens}
 \end{aligned}$$

Équation récursive :

$$\begin{aligned}
 \text{ensval}(a) &= g_1(a, \text{ensval}_F(\text{fils}(a))) \\
 \text{ensval}_F(s) &= g_2(s, \text{ensval}(\text{premier}(s)), \text{ensval}_F(\text{reste}(s)))
 \end{aligned}$$

On pose :

$$g_1(a, z) = \{\text{racine}(a)\} \cup z \quad g_2(a, z_1, z_2) = z_1 \cup z_2$$

Ceci donne :

$$\begin{aligned}
 \text{ensval}(a) &= \emptyset && \text{si est\_arbre\_vider}(a) \\
 \text{ensval}(a) &= \{\text{racine}(a)\} \cup \text{ensval}_F(\text{fils}(a)) && \text{sinon} \\
 \\ 
 \text{ensval}_F(s) &= \emptyset && \text{si liste\_vider}(s) \\
 \text{ensval}_F(s) &= \text{ensval}(\text{premier}(s)) \cup \text{ensval}_F(\text{reste}(s)) && \text{sinon}
 \end{aligned}$$

**Exercice :** implanter la fonction `ensval` en OCAML.

### Les arbres binaires de recherche (ABR)

Un arbre binaire de recherche est un arbre binaire de type I ayant des étiquettes **distinctes** de type **entier** t.q.

- Le sous-arbre gauche de *tout* nœud  $n$  ne contient que des entiers inférieurs ou égaux à  $n$ ;
- Le sous-arbre droit de *tout* nœud  $n$  ne contient que des entiers strictement supérieurs à  $n$ ;

### Quelques exemples



11

### Recherche d'un élément dans un ABR

Pour rechercher un élément dans un ABR  $a$  on doit :

1. Comparer avec la racine de  $a$ ,
2. Rechercher dans le fils droit ou gauche de  $a$ .

```

# let rec recherche(e,a) = match a with
  Av -> false
  | Abr(r,fg,fd) -> if e=r then true
                    else if e<r then recherche(e,fg)
                        else recherche(e,fd);;
val recherche : 'a * 'a abin -> bool = <fun>

```

### Complexité de la recherche dans un ABR

Considérons un ABR avec  $n$  éléments.

La complexité dans le meilleur des cas est donc  $O(1)$ .

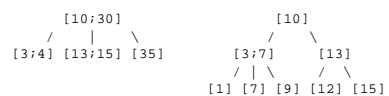
La complexité au pire est donc en  $O(n)$ .

### Les arbres 2-3-4

Un arbre 2.3.4 est un arbre 4-aire étiqueté de type II t.q.

1. chaque nœud est étiqueté par une liste de  $1 \leq k \leq 3$  d'éléments  $[x_1, \dots, x_k]$  t.q.  $x_1 \leq \dots \leq x_k$
2. un nœud étiqueté par  $[x_1, \dots, x_k]$  contient  $k+1$  fils  $f_1, \dots, f_{k+1}$  t.q.
  - tous les éléments de  $f_i$  sont inférieurs ou égaux à  $x_i$  (pour  $1 \leq i \leq k$ )
  - tous les éléments de  $f_i$  sont strictement supérieurs à  $x_{i-1}$  (pour  $2 \leq i \leq k+1$ )
3. toutes les feuilles sont au même niveau

### Quelques exemples



12

### Recherche d'un élément dans un arbre 2.3.4

Pour rechercher un élément dans un arbre 2.3.4  $a$  on doit :

1. Comparer avec les éléments de la racine de  $a$ ,
2. Rechercher dans le "bon" fils selon la première comparaison.

Exercice : implanter la fonction `recherche` en OCAML.

### Complexité de la recherche dans un arbre 2.3.4

Considérons un arbre 2.3.4  $a$  avec  $n$  éléments et hauteur  $h(a)$ .

- Une borne inférieure : si l'arbre 2.3.4  $a$  contient uniquement un élément dans chaque noeud, alors  $2^{h(a)} - 1 \leq n$
- Une borne supérieure : si l'arbre 2.3.4  $a$  contient trois éléments dans chaque noeud, alors  $n \leq 4^{h(a)} - 1$

$$2^{h(a)} - 1 \leq n \leq 4^{h(a)} - 1$$

$$2^{h(a)} \leq n + 1 \leq 4^{h(a)}$$

$$\log_4(n + 1) \leq h(a) \leq \log_2(n + 1)$$

La complexité dans le meilleur des cas est donc  $O(1)$ .

La complexité au pire est donc en  $O(\log(n + 1))$ .

### Preuve de propriétés par récurrence sur les arbres

- On utilise la définition inductive de l'ensemble d'arbres
- On utilise la définition récursive d'une ou plusieurs fonctions et/ou propriétés.

### Exemple I

Propriété à démontrer :  $\forall a \text{ abin } b, \text{miroir}(\text{miroir}(b)) = b$

Preuve : Par induction sur  $b$

- Cas de base :  $b = \text{avide}$ .

$$\begin{aligned} \text{miroir}(\text{miroir}(\text{avide})) &= \\ \text{miroir}(\text{avide}) &= \\ \text{avide} & \end{aligned}$$

- Cas inductif :  $b = \text{abr}(m, a_1, a_2)$ .

$$\begin{aligned} \text{miroir}(\text{miroir}(b)) &= \\ \text{miroir}(\text{miroir}(\text{abr}(m, a_1, a_2))) &= \\ \text{miroir}(\text{abr}(m, \text{miroir}(a_2), \text{miroir}(a_1))) &= \\ \text{abr}(m, \text{miroir}(\text{miroir}(a_1)), \text{miroir}(\text{miroir}(a_2))) &=_{h,r} \\ \text{abr}(m, a_1, a_2) &= \\ b & \end{aligned}$$

### Exemple II

Propriété à démontrer :  $\forall \text{AbinSE } a, \text{peigne}(b) \rightarrow \text{prof}(b) = \text{nb\_internes}(b)$

Où `peigne` est la propriété :

$$\text{peigne}(b) = \text{peigne\_gauche}(b) \vee \text{peigne\_droit}(b)$$

$$\begin{aligned} \text{peigne\_gauche}(\text{avide}) &= \text{true} \\ \text{peigne\_gauche}(\text{noeud}(a_1, a_2)) &= \text{peigne\_gauche}(a_1) \wedge \\ &\quad \text{est\_arbre\_vide}(a_2) \\ \text{peigne\_droit}(\dots) &= \dots \end{aligned}$$

Preuve : Par induction sur  $b$

- Cas de base :  $b = \text{avide}$ .

$$\text{prof}(\text{avide}) = 0 = \text{nb\_internes}(\text{avide})$$

- Cas inductif :  $b = \text{noeud}(a_1, a_2)$ .

$\text{peigne}(\text{noeud}(a_1, a_2))$  implique

$$\text{peigne\_gauche}(\text{noeud}(a_1, a_2)) \vee \text{peigne\_droit}(\text{noeud}(a_1, a_2))$$

Supposons sans perte de généralité

$$\text{peigne\_gauche}(\text{noeud}(a_1, a_2))$$

C'est à dire :  $\text{peigne\_gauche}(a_1) \wedge a_2 = \text{avide}$

Alors :  $\text{prof}(\text{noeud}(a_1, a_2)) = 1 + \text{prof}(a_1)$  et

$\text{nb\_internes}(\text{noeud}(a_1, a_2)) = 1 + \text{nb\_internes}(a_1) + 0$

Comme  $\text{peigne\_gauche}(a_1)$ , alors par h.r.  $\text{prof}(a_1) = \text{prof}(a_1) =$

$\text{nb\_internes}(a_1)$  et donc

$\text{prof}(b) = 1 + \text{prof}(a_1) = 1 + \text{nb\_internes}(a_1) = \text{nb\_internes}(b)$ .