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.

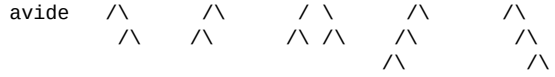
Arbre binaire (II) : tout nœud interne a au plus deux fils.

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.

- $avide \in AbinSE$
- Si $a_1, a_2 \in AbinSE$, alors $noeud(a_1, a_2) \in AbinSE$.

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 arb1 = N(N(Av,Av),Av);;
val arb1 : abinse = N (N (Av, Av), Av)

# est_arbre_vide arb1;;
- : bool = false
```

```
# arbre_gauche arb1 ;;
- : abinse = N (Av, Av)

# arbre_droite arb1 ;;
- : abinse = Av
```

Équations récursives pour AbinSE

Schéma récursif :

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

Exemple : nombre de feuilles

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

Déclaration du type :

$$\texttt{nb\_f} : \texttt{AbinSE} \rightarrow \texttt{entier}$$

Équation récursive :

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

Cas particulier :

a=avide

On pose

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

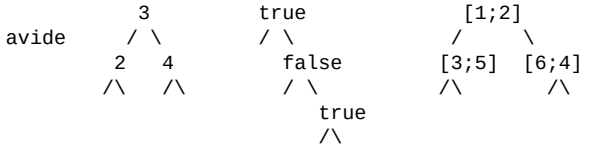
Ceci donne :

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

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>
```


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



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

- $avide \in$ 'a abin
- Si $a_1, a_2 \in$ 'a abin et m est de type 'a, alors $abr(m, a_1, a_2) \in$ '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

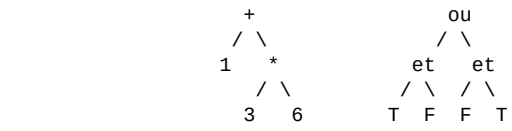
fils_droit : 'a abin → 'a abin

'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>
```

```
# 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_vide a = match a with
  Av -> true | _ -> false;;
val est_arbre_vide : 'a abin -> bool = <fun>

# let arb1 = Arb(4,Arb(2,Av,Av),Av);;
val arb1 : 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 arb1;;
- : 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.

- Chaque nœud est unaire ou binaire.

Les arbres binaires de type II avec étiquette

C'est un ensemble parametré 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 $feuille(m) \in ('a,'b)$ `AbinII`
- Si $a_1, a_2 \in ('a,'b)$ `AbinII` et o est de type `'b`, alors $noeud(o, a_1, a_2) \in ('a,'b)$ `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";;
```

```
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 arb1 = N(Mul,F(4),F(5));;
val arb1 : (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

```
avide      3
          / \
             5
```



```
           / | \
          1  7
          |  |\
             6  8
             |  |
```

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.
- $avide \in$ `’a arbre`
- Si $m \in$ `’a` et $l \in$ `(’a arbre) liste`, alors $noeud(m, l) \in$ `’a arbre`.

Un exemple

```
      3            noeud(3,
     / \                 [avide,
        5                 noeud(5,
       / | \                    [noeud(1,[]),
      1  7                      noeud(7,
      |  |\                           [noeud(6,[]),
         6  8                         noeud(8,[])])
         |  |                    avide])])
```

Type abstrait pour un arbre quelconque

Domaine :
`’a arbre`
Opérations de construction :
`avide` : `’a arbre`
`cons_arbre` : `’a` $\times$ `(’a arbre) liste` $\rightarrow$ `’a arbre`
Opérations de test :
`est_arbre_vide` : `’a arbre` $\rightarrow$ `booléen`
Opérations d’accès :
`racine` : `’a arbre` $\rightarrow$ `’a`
`fils` : `’a arbre` $\rightarrow$ `(’a arbre) liste`

En OCAML

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

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

(* opération de construction *)
# let cons_arbre_vide = Av;;
val cons_arbre_vide : '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_vide a = match a with
  Av -> true | _ -> false;;
val est_arbre_vide : '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 :

$$\begin{array}{rcl} \texttt{f}(a) & = & g_1(a, \texttt{f\_F}(\texttt{fils}(a)) \\ \texttt{f\_F}(s) & = & g_2(s, \texttt{f}(\texttt{premier}(s)), \texttt{f\_F}(\texttt{reste}(s))) \end{array}$$

Le domaine de `f` est `'a arbre`

Le domaine de `f_F` est `('a arbre) liste`

Cas particuliers :

$a = \texttt{avide}$ (`fils(avide)` n'est pas défini).

$s = \texttt{nil}$ (`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 :

$$\texttt{taille} : \texttt{'a arbre} \rightarrow \texttt{entier}$$
$$\texttt{taille\_F} : \texttt{('a arbre) liste} \rightarrow \texttt{entier}$$

Équation récursive :

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

On pose

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

Ceci donne :

$$\begin{array}{rcll} \texttt{taille}(a) & = & 0 & \\ & & & \text{si } \texttt{est\_arbre\_vide}(a) \\ \texttt{taille}(a) & = & 1 + \texttt{taille\_F}(\texttt{fils}(a)) & \\ & & & \text{sinon} \end{array}$$

$$\begin{array}{rcll} \texttt{taille\_F}(s) & = & 0 & \\ & & & \text{si } \texttt{liste\_vide}(s) \\ \texttt{taille\_F}(s) & = & \texttt{taille}(\texttt{premier}(s)) + \texttt{taille\_F}(\texttt{reste}(s)) & \\ & & & \text{sinon} \end{array}$$

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 :

$$\texttt{ensval} : \texttt{'a arbre} \to \texttt{'a ens}$$
$$\texttt{ensval\_F} : \texttt{('a arbre) liste} \to \texttt{'a ens}$$

Équation récursive :

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

On pose :

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

Ceci donne :

$$\begin{array}{lcl} \texttt{ensval}(a) & = & \emptyset \\ & & \text{si } \texttt{est\_arbre\_vide}(a) \\ \texttt{ensval}(a) & = & \{\texttt{racine}(a)\} \cup \texttt{ensval\_F}(\texttt{fils}(a)) \\ & & \text{sinon} \end{array}$$

$$\begin{array}{lcl} \texttt{ensval\_F}(s) & = & \emptyset \\ & & \text{si } \texttt{liste\_vide}(s) \\ \texttt{ensval\_F}(s) & = & \texttt{ensval}(\texttt{premier}(s)) \cup \texttt{ensval\_F}(\texttt{reste}(s)) \\ & & \text{sinon} \end{array}$$

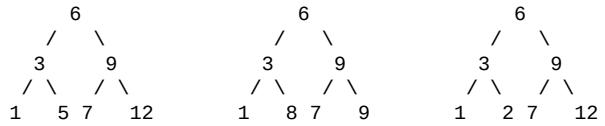
Exercice : implanter la fonction `ensval` en OCAML.

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



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

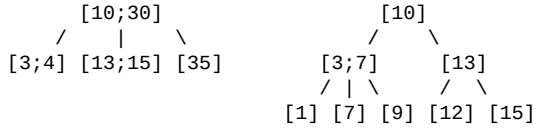
Considérons un ABR avec n éléments.
La complexité dans le meilleur des cas est donc $O(1)$.
La complexité au pire est donc en $O(n)$.

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 $[x_1, \ldots, x_k]$ t.q. $x_1 \leq \ldots \leq x_k$
2. un nœud étiqueté par $[x_1, \ldots, x_k]$ contient $k+1$ fils $f_1, \ldots, f_{k+1}$ 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



Recherche d'un élément dans un arbre 2.3.4

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

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

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

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

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 plusieures fonctions et/ou propriétés.

Exemple I

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

Preuve : Par induction sur b

- Cas de base : $b = avide$.

$$\begin{array}{ll} \mathtt{miroir}(\mathtt{miroir}(avide)) & = \\ \mathtt{miroir}(avide) & = \\ avide & \end{array}$$

- Cas inductif : $b = abr(m, a_1, a_2)$.

$$\begin{array}{ll} \mathtt{miroir}(\mathtt{miroir}(b)) & = \\ \mathtt{miroir}(\mathtt{miroir}(abr(m, a_1, a_2))) & = \\ \mathtt{miroir}(abr(m, \mathtt{miroir}(a_2), \mathtt{miroir}(a_1))) & = \\ abr(m, \mathtt{miroir}(\mathtt{miroir}(a_1), \mathtt{miroir}(\mathtt{miroir}(a_2)))) & =_{h.r} \\ abr(m, a_1, a_2) & = \\ b & \end{array}$$

Exemple II

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

Où `peigne` est la propriété :

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

$$\begin{array}{ll} \mathtt{peigne\_gauche}(avide) & = true \\ \mathtt{peigne\_gauche}(noeud(a_1, a_2)) & = \mathtt{peigne\_gauche}(a_1) \wedge \\ & \mathtt{est\_arbre\_vide}(a_2) \\ \mathtt{peigne\_droit}(\ldots) & = \ldots \end{array}$$

Preuve : Par induction sur b

- Cas de base : $b = avide$.

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

- Cas inductif : $b = noeud(a_1, a_2)$.

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

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

Supposons sans perte de généralité

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

C'est à dire : $\mathtt{peigne\_gauche}(a_1) \wedge a_2 = avide$
Alors : $\mathtt{prof}(noeud(a_1, a_2)) = 1 + \mathtt{prof}(a_1)$ et
$\mathtt{nb\_internes}(noeud(a_1, a_2)) = 1 + \mathtt{nb\_internes}(a_1) + 0$
Comme $\mathtt{peigne\_gauche}(a_1)$, alors par h.r. $\mathtt{prof}(a_1) = \mathtt{prof}(a_1) = \mathtt{nb\_internes}(a_1)$ et donc
$\mathtt{prof}(b) = 1 + \mathtt{prof}(a_1) = 1 + \mathtt{nb\_internes}(a_1) = \mathtt{nb\_internes}(b)$.