

Algorithmique et Approche Fonctionnelle

Cours 8 : Les arbres

3 Novembre 2008

La structure d'arbre est très utilisée en informatique, que ce soit pour :

- présenter un ensemble d'objets ou d'informations élémentaires organisé en une structure hiérarchique
- en faciliter l'accès ou la recherche
- modéliser de nombreux problèmes

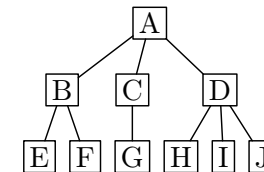
les structures d'arbres constituent un complément indispensable de la structure de liste

- liste = organisation **linéaire** de l'information où tous les objets sont au **même niveau**
- arbre = organisation **hiérarchique** de l'information en **plusieurs niveaux**

exemples de hiérarchies

- le sommaire d'un **livre** reflète une hiérarchie
 - I. chapitres
 - 1. sections
 - a. sous-sections
 - paragraphes
- l'organisation des **fichiers** d'un système d'exploitation en répertoires et sous-répertoires correspond à une structure d'arbre
- on utilise des **arbres de classifications** pour classer des éléments (ex. instruments de musique)

terminologie



- **A** est la **racine** de l'arbre
- **A, B, ..., J** sont les **nœuds** de l'arbre
- **A** a pour **fil** **B, C** et **D**
- **E, F, ..., J** sont des **feuilles**, *i.e.* des nœuds sans fils
- **A, B, C** et **D** sont des **nœuds internes**
- **A** a trois **sous-arbres**
- **ACG** est une **branche**

- les **arbres binaires** sont des arbres tels que tout nœud a au plus **deux** fils
- ces arbres sont simples à mettre en œuvre et utilisés dans de nombreuses modélisations
- on peut donner une définition récursive d'un arbre binaire :
 - un **arbre** est une **feuille**
 - ou un **arbre** est un nœud interne qui peut être décomposé en la racine et les **fils gauche et droit**, qui sont également des **arbres**
- on peut de plus associer une **valeur** à chaque feuille et chaque nœud interne

la notion d'**arbre vide** est utilisée pour uniformiser la représentation des arbres binaires, elle permet de ne pas distinguer les feuilles des nœuds internes

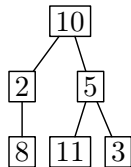
- un arbre vide se comporte comme une feuille à laquelle il n'est pas associé de valeur
- une feuille avec valeur peut alors se représenter comme un nœud interne dont tous les fils sont des arbres vides
- lorsqu'un arbre est non vide, il est caractérisé par sa racine à laquelle est associée une valeur et ses fils, qui sont **eux-mêmes des arbres**

arbres binaires en OCAML

le type arbre suivant permet de définir des **arbres binaires** dont les feuilles et les nœuds contiennent des **entiers**

```
# type arbre = Vide | Noeud of int * arbre * arbre;;
```

```
type arbre = Vide | Noeud of int * arbre * arbre
```



```
# let a =
  Noeud(10,
    Noeud(2, Noeud(8, Vide, Vide), Vide),
    Noeud(5, Noeud(11, Vide, Vide), Noeud(3, Vide, Vide)));;
val a : arbre = Noeud (10, ... , ...)
```

arbres binaires polymorphes

on définit une structure d'arbre binaire **polymorphe** en ajoutant un paramètre de type

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

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

```
let b = Noeud(10, Noeud(5, Vide, Vide), Vide);;
```

```
val b : int arbre = Noeud (10, Noeud (5, Vide, Vide), Vide)
```

```
# let c = Noeud('f', Vide, Noeud('a', Vide, Noeud('g', Vide, Vide)));;
```

```
val c : char arbre =
  Noeud ('f', Vide, Noeud ('a', Vide, Noeud ('g', Vide, Vide)))
```

taille d'un arbre binaire

la fonction `taille` renvoie le **nombre de nœuds** d'un arbre binaire

```
# let rec taille a =  
  match a with  
  | Vide -> 0  
  | Noeud(_,g,d) -> 1 + taille g + taille d;;  
val taille : 'a arbre -> int = <fun>
```

```
# taille a;;  
- : int = 6
```

```
# taille b;;  
- : int = 2
```

profondeur d'un arbre binaire

la fonction `profondeur` renvoie la longueur de la plus grande branche d'un arbre binaire

```
# let rec profondeur a =  
  match a with  
  | Vide -> 0  
  | Noeud(_,g,d) -> 1 + max (profondeur g) (profondeur d);;  
val profondeur : 'a arbre -> int = <fun>
```

```
# profondeur b;;
```

```
- : int = 2
```

```
# profondeur c;;
```

```
- : int = 3
```

relation en taille et profondeur

- $\text{profondeur} \leq \text{taille}$
- $\text{taille} \leq 2^{\text{profondeur}} - 1$
- un arbre binaire est dit **complet** si : $\text{taille} = 2^{\text{profondeur}} - 1$

miroir d'un arbre binaire

la fonction `miroir` retourne l'image miroir d'un arbre binaire

```
# let rec miroir a =  
  match a with  
  | Vide -> Vide  
  | Noeud(r,g,d) -> Noeud(r,miroir d,miroir g);;  
val miroir : 'a arbre -> 'a arbre = <fun>
```

```
# miroir a;;
```

```
- : int arbre =
```

```
Noeud (10,  
  Noeud (5, Noeud (3, Vide, Vide), Noeud (11, Vide, Vide)),  
  Noeud (2, Vide, Noeud (8, Vide, Vide)))
```

de nombreuses fonctions sur les arbres binaires consistent à les parcourir suivant un certain ordre

- **préfixe** : on traite d'abord la **racine**, puis on parcourt le sous-arbre **gauche**, et enfin le sous-arbre **droit**
- **infixe** : on parcourt d'abord le sous-arbre **gauche**, puis on traite la **racine**, et enfin le sous-arbre **droit**
- **suffixe** : on parcourt d'abord le sous-arbre **gauche**, puis le sous-arbre **droit**, et enfin la **racine**

comme pour les listes, on peut définir des itérateurs sur les arbres binaires

la fonction **fold_gdr** réalise par exemple un parcours **suffixe** d'un arbre binaire en appliquant une fonction **f** à la racine et aux résultats des sous-arbres gauche et droit

```
# let rec fold_gdr f acc a =
  match a with
  | Vide -> acc
  | Noeud(r,g,d) ->
    let vg = fold_gdr f acc g in
    let vd = fold_gdr f acc d in
    f r vg vd;;

val fold_gdr :
  ('a -> 'b -> 'b -> 'b) -> 'b -> 'a arbre -> 'b = <fun>
```

exemples

les fonctions **taille**, **profondeur** et **miroir** peuvent être réécrites en utilisant l'itérateur **fold_gdr**

```
# let taille = fold_gdr (fun _ x y -> 1+x+y) 0;;
val taille : 'a arbre -> int = <fun>
```

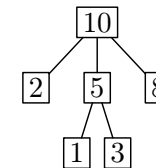
```
# let profondeur = fold_gdr (fun _ x y -> 1 + max x y) 0;;
val profondeur : 'a arbre -> int = <fun>
```

```
# let miroir = fold_gdr (fun r x y -> Noeud(r,y,x)) Vide;;
val miroir : 'a arbre -> 'a arbre = <fun>
```

arbres n-aires

- les arbres **n-aires** ont un nombre de sous-arbres arbitraire
- on représente les arbres n-aires polymorphes à l'aide du type suivant

```
# type 'a arbre = Vide | Noeud of 'a * 'a arbre list;;
type 'a arbre = Vide | Noeud of 'a * 'a arbre list
```



```
# let a =
  Noeud(10, [ Noeud(2, []);
              Noeud(5, [Noeud(11, []); Noeud(3, [])]);
              Noeud(8, []) ]);
val a : int arbre = Noeud (10, [...])
```

les fonctions **taille** et **profondeur** pour les arbres n-aires se définissent de la manière suivante

```
# let rec taille a =
  match a with
  | Vide -> 0
  | Noeud(_,l) ->
    1 + List.fold_left (fun acc x -> acc + taille x) 0 l;;
val taille : 'a arbre -> int = <fun>
```

```
# let rec profondeur a =
  match a with
  | Vide -> 0
  | Noeud(_,l) ->
    1 + List.fold_left
      (fun acc x -> max acc (profondeur x)) 0 l;;
val profondeur : 'a arbre -> int = <fun>
```

la fonction **liste_arbre** retourne une liste formée des éléments d'un arbre n-aire

```
# let list_arbre a =
  let rec liste_rec acc a =
    match a with
    | Vide -> acc
    | Noeud(r,l) -> List.fold_left liste_rec (r::acc) l
  in liste_rec [] a;;
val liste_arbre : 'a arbre -> 'a list = <fun>
```

```
# liste_arbre a;;
- : int list = [10 ; 2 ; 5 ; 11 ; 3 ; 8]
```