Programmation fonctionnelle

Notes de cours

Cours 9

23 novembre 2011

Sylvain Conchon

sylvain.conchon@lri.fr

Les notions abordées cette semaine

- Les foncteurs `Set.Make` et `Map.Make` d'Ocaml
- Arbres binaires
- Arbres binaires de recherche
- Arbres équilibrés : AVL

Le foncteur Set.Make

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module type S = sig
  type elt
  type t
  val empty : t
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val union : t -> t -> t
  val compare : t -> t -> int
  ...
end

module Make (Ord : OrderedType) :S with type elt = Ord.t
```

Utilisation de Set.Make

- Des ensembles d'entiers

```
module Int = struct
  type t = int
  let compare = Pervasives.compare
end

module S = Set.Make(Int)

let s1 = S.add 1 (S.add 2 S.empty)
let s2 = S.add 3 (S.add 4 S.empty)
let s3 = S.union s1 (S.remove 2 s1)
```

- Des ensembles d'ensembles d'entiers

```
module P = Set.Make(S)

let p = P.add s1 (P.add s3 P.empty)
```

Le foncteur Map.Make

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module type S = sig
  type key
  type (+'a) t

  val empty : t
  val add : key -> 'a -> 'a t -> 'a t
  val remove : key -> 'a t -> 'a t
  val find : key -> 'a t -> 'a
  ...
end

module Make (Ord : OrderedType) :S with type key = Ord.t
```

Arbres binaires

Arbres binaires

Les arbres binaires avec des informations de type 'a aux nœuds sont définis avec le type suivant :

```
type 'a arbre =
   Vide | Noeud of 'a * 'a arbre * 'a 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, ... , ...)
```

Recherche dans un arbre binaire

La fonction `recherche`, de type `'a -> 'a arbre -> bool` recherche un élément dans un arbre binaire :

```
let rec recherche e = function
  | Vide -> false
  | Noeud(x,g,d) ->
      x=e || recherche e g || recherche e d
```

- Le temps de recherche dans le pire des cas est en $O(n)$.
- Il faut des hypothèses plus fortes sur la structure de l'arbre afin d'obtenir une meilleure complexité.

Arbre binaire de recherche

Arbres ordonnés (ou de recherche)

Un arbre binaire est ordonné (ou de recherche) par rapport à une relation d'ordre quelconque si :

- c'est l'arbre `Vide`
- ou c'est un arbre non-vide `Noeud(x,g,d)` et
  1. les éléments du sous-arbre gauche `g` sont inférieurs à la racine `x`
  2. la valeur `x` stockée à la racine de l'arbre est inférieure aux éléments du sous-arbre droit `d`
  3. les sous-arbres `g` et `d` sont eux-mêmes ordonnés

Recherche d'un élément

La structure ordonnée des arbres binaires de recherche permet d'effectuer la recherche d'un élément avec une compléxité en moyenne de $O(\log n)$.

```
let rec recherche e = function
  | Vide -> false
  | Noeud (x, _, _) when x=e -> true
  | Noeud (x, g, _) when e<x -> recherche e g
  | Noeud (_, _, d) -> recherche e d
```

Ajout d'un élément

L'ajout d'un élément dans un arbre binaire de recherche peut se faire de deux manières :

- ajout aux feuilles : facile à définir
- ajout à la racine : utile si les recherches portent sur les éléments récemment ajoutés

Ajout aux feuilles

La fonction `ajout : 'a -> 'a arbre -> 'a arbre` est définie de la façon suivante :

```
let rec ajout e a =
 match a with
  | Vide -> Noeud(e,Vide,Vide)
  | Noeud(x, _, _) when e=x -> a
  | Noeud(x, g, d) when x<e -> Noeud(x, g, ajout e d)
  | Noeud(x, g, d) -> Noeud(x, ajout e g, d)
```

Ajout à la racine

L'ajout à la racine d'un élément `x` consiste à

- "couper" un arbre en deux sous-arbres (de recherche) `g` et `d`, contenant respectivement les éléments plus petits et plus grands que `x`
- construire l'arbre `Noeud(x,g,d)`

Couper un arbre en deux

La fonction `coupe : 'a -> 'a arbre -> 'a arbre * 'a arbre` réalise la coupure d'un arbre.

```
let rec coupe e a =
 match a with
  | Vide -> (Vide , Vide)
  | Noeud(x, g, d) when x=e -> (g , d)
  | Noeud(x, g, d) when x<e ->
    let (t1 , t2) = coupe e d in
    (Noeud(x, g, t1), t2)
  | Noeud(x, g, d) ->
    let (t1 , t2) = coupe e g in
    (t1 , Noeud(x, t2, ld))
```

Ajout à la racine

La fonction `ajout : 'a -> 'a arbre -> 'a arbre` est alors définie de la manière suivante :

```
let ajout e a =
  let (g , d) = coupe e a in Noeud(e,g,d)
```

Suppression d'un élément (1/2)

La suppression d'un élément `x` dans un arbre binaire de recherche consiste à :

- isoler le sous-arbre `Noeud(x, g, d)`
- supprimer le plus grand élément `y` de `g` (on obtient ainsi un arbre de recherche `h`)
- reconstruire l'arbre `Noeud(y, h, d)`

```
let rec enleve_plus_grand = function
  | Vide -> raise Not_found
  | Noeud(x, g, Vide) -> (x, g)
  | Noeud(x, g, d) ->
    let (y, d') = enleve_plus_grand d in
    (y, Noeud(x, g, d'))
```

```
val enleve_plus_grand : 'a arbre -> 'a * 'a arbre
```

Suppression d'un élément (2/2)

La fonction `suppression : 'a -> 'a arbre -> 'a arbre` est alors définie par :

```
let rec suppression e a =
 match a with
 | Vide -> Vide
 | Noeud(x, Vide, d) when x=e -> d
 | Noeud(x, g, d) when x=e ->
    let (y, g') = enleve_plus_grand g in
    Noeud(y,g',d)
 | Noeud(x, g, d) when e<x ->
    Noeud(x, suppression e g,d)
 | Noeud(x, g, d) ->
    Noeud(x, g, suppression e d)
```

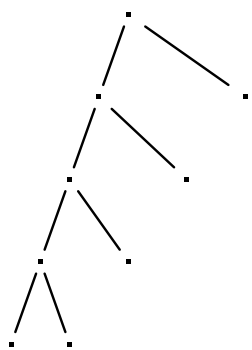
Les arbres équilibrés

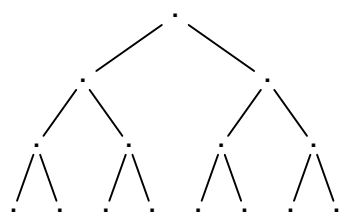
Recherche de l'équilibre

- La recherche dans un arbre ordonné est proportionnelle à la longueur de la plus grande branche de l'arbre
- Cette recherche est optimale pour des arbres de recherche équilibrés, c'est-à-dire les arbres de taille n et de hauteur $\log(n)$

Rééquilibrage des arbres de recherche

- L'efficacité de la recherche dans un arbre binaire ordonné dépend fortement de la forme de l'arbre
- La forme la pire est celle du peigne : un arbre où chaque nœud n'a qu'un seul successeur gauche ou droit (l'arbre n'est alors ni plus ni moins qu'une liste) ; l'opération de recherche est alors en $O(n)$
- La forme la meilleure est celle de l'arbre "équilibré", i.e. où $\mathit{taille} \approx 2^{\mathit{prof}}$ soit $\mathit{prof} \approx \log_2(\mathit{taille})$





Il faut donc essayer de maintenir l'équilibre des arbres binaires de recherche dans les opérations d'ajout, de suppression etc.

Les arbres AVL

- Premiers arbres binaires équilibrés
- Inventés en 1962 par les russes Adelson-Velsky et Landis (d'où le nom AVL)
- Ces arbres vérifient la propriété suivante :

  La différence entre les hauteurs des fils gauche et des fils droit de tout nœud ne peut excéder 1

Définition des arbres AVL

- Le type des AVL est similaire à celui des arbres binaires
- On ajoute un entier dans les noeuds afin de mémoriser la hauteur des arbres

```
type 'a avl =
  Vide | Noeud of 'a * 'a avl * 'a avl * int
```

On manipule les hauteurs des arbres à l'aide des deux fonctions suivantes :

```
let height = function
  | Vide -> 0
  | Noeud(_, _, _, h) -> h

let creation l v r =
  Noeud(v, l, r, 1 + max (height l) (height r))
```

Ajout d'un élément

```
let rec ajout x = function
  | Vide ->
     creation Vide x Vide
  | Noeud(v, l, r, _) as t ->
     if x = v then t else
     if x < v then
      equilibrage (ajout x l) v r
     else
      equilibrage l v (ajout x r)
```

Équilibrage

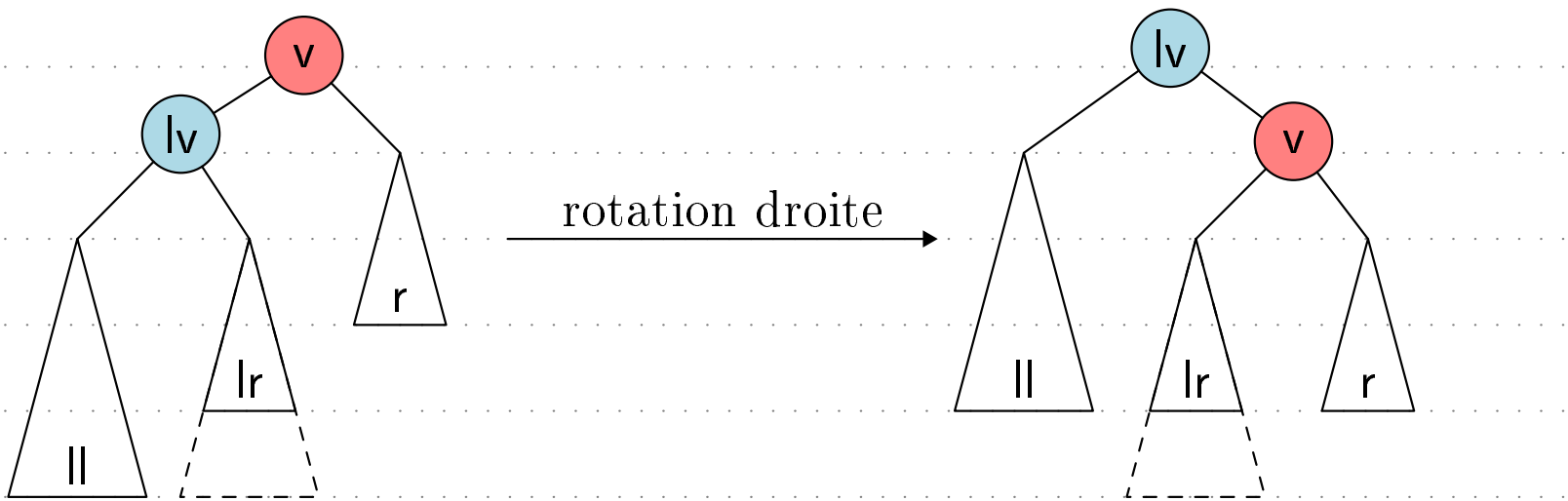
Soient deux arbres `l` et `r` équilibrés tels que $|\texttt{prof}(l) - \texttt{prof}(g)| \leq 2$
La fonction `equilibrage` construit un arbre équilibré formé des mêmes éléments, et dans le même ordre, que `Noeud(l,v,r)`.

```
let equilibrage l v r =
 let (hl, hr) = (height l, height r) in
 if hl > hr + 1 then begin
  match l with
   | Noeud(lv, ll, lr, _) when height ll >= height lr ->
     creation ll lv (creation lr v r)
   | Noeud(lv, ll, Noeud(lrv, lrl, lrr, _), _)->
     creation (creation ll lv lrl) lrv (creation lrr v r)
   | _ -> assert false
  end else if hr > hl + 1 then begin
    (* cas symétrique *)
 end else creation l v r
```

Équilibrage à droite par simples rotations

Si $\texttt{hl} = \texttt{hr} + 2$ et $\texttt{height(lr)} \leq \texttt{height(ll)} \leq \texttt{height(lr)} + 1$
alors on réalise une rotation simple à droite :

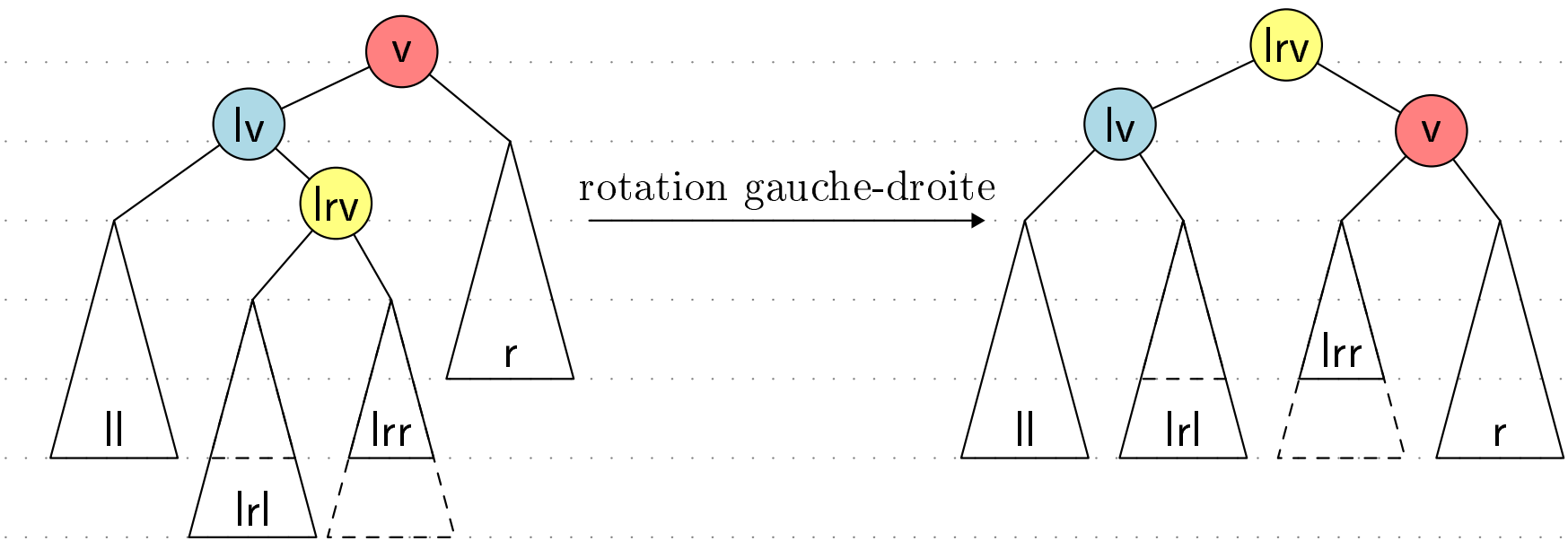
```
creation ll lv (creation lr v r)
```



Équilibrage à droite par doubles rotations

Si $\texttt{hl} = \texttt{hr} + 2$ et $\texttt{height}(\texttt{ll}) = p$ et $\texttt{height}(\texttt{lr}) = p + 1$ alors on réalise une rotation double gauche-droite :

```
creation (creation ll lv lrl) lrv (creation lrr v r)
```



Suppression d'un élément

La suppression d'un élément `x` dans un AVL `Noeud(v, l, r, h)` consiste à :

- Fusionner `l` et `r` si `x=v`
- Supprimer `x` dans `l` ou `r` selon la valeur de `x` $<$ `v`
- Re-équilibrer l'arbre obtenu

```
let rec suppression x = function
  | Vide -> Vide
  | Noeud(v, l, r, _) ->
    if x = v then fusion l r else
    if x < v then equilibrage (suppression x l) v r
    else equilibrage l v (suppression x r)
```

Fusion de deux AVLs

Soient deux AVLs `t1` et `t2` tels que $|\texttt{heigth}(t1) - \texttt{height}(t2)| \leq 1$.

- Tous les éléments de `t1` sont plus petits que ceux de `t2`
- La fonction `fusion` construit un AVL formé des mêmes éléments que `t1` et `t2`

```
let fusion t1 t2 =
 match (t1, t2) with
  | (Vide, t) | (t, Vide) -> t
  | (_, _) ->
      equilibrage t1 (min_elt t2) (suppr_min_elt t2)
```

Plus petit élément

La fonction `min_elt` retourne le plus petit élément d'un arbre binaire de recherche.

```
let rec min_elt = function
  | Vide -> raise Not_found
  | Noeud(v, Vide, r, _) -> v
  | Noeud(v, l, r, _) -> min_elt l
```

La fonction `suppr_min_elt` supprime le plus petit élément d'un AVL

```
let rec suppr_min_elt = function
  | Vide -> raise Not_found
  | Noeud(v, Vide, r, _) -> r
  | Noeud(v, l, r, _) ->
     equilibrage (suppr_min_elt l) v r
```