Algorithmique et Approche Fonctionnelle

Cours 9 : Arbres de recherche

17 Novembre 2008

Arbre binaire de recherche

recherche dans un arbre binaire

la fonction suivante permet de rechercher 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
val recherche : 'a -> 'a arbre -> bool = <fun>
```

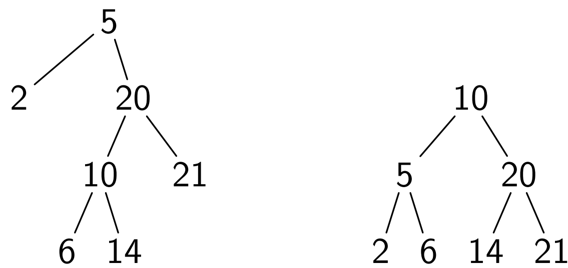
- le temps de recherche est proportionnel à la taille de l'arbre ($O(n)$)
- il faut des hypothèses plus fortes sur la structure de l'arbre afin d'obtenir une meilleure complexité

arbres ordonnés

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

- c'est l'arbre vide (`Vide`)
- 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 racine `x` est inférieure aux éléments du sous-arbre droit `d`
  3. les sous-arbres `g` et `d` sont eux-mêmes ordonnés

exemples d'arbres ordonnés




```
let a1 =
 Noeud(5,Noeud(2,Vide,Vide),
  Noeud(20,Noeud(10,Noeud(6,Vide,Vide),Noeud(14,Vide,Vide)),
           Noeud(21,Vide,Vide)))
```

```
let a2 =
 Noeud(10,
  Noeud(5,Noeud(2,Vide,Vide),Noeud(6,Vide,Vide)),
  Noeud(20,Noeud(14,Vide,Vide),Noeud(21,Vide,Vide)))
```


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
val recherche : 'a -> 'a arbre -> bool = <fun>
```


évaluation de la fonction `recherche`

évaluation de `recherche 10 a1`

		`recherche 10 Noeud(5,...)`
$5 < 10$	$\Rightarrow$	`recherche 10 Noeud(20,...)`
$20 > 10$	$\Rightarrow$	`recherche 10 Noeud(10,...)`
$10 = 10$	$\Rightarrow$	`true`

ajout d'un élément

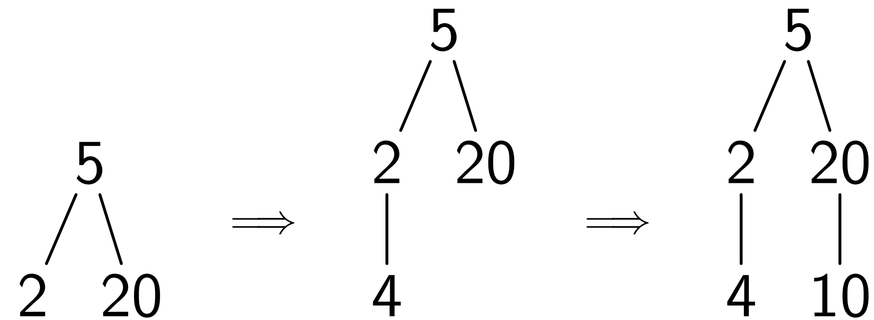
l'ajout d'un élément dans un arbre binaire de recherche peut se faire de deux manières

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

ajout aux feuilles

```
# 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)
val ajout : 'a -> 'a arbre -> 'a arbre = <fun>
```


exemple d'ajout aux feuilles



ajout à la racine (1/2)

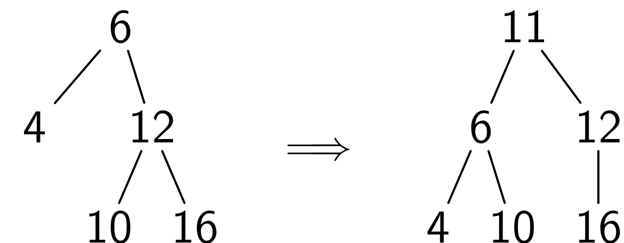
l'ajout à la racine d'un élément `x` consiste à

- "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)`

```
# 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,d))
val coupe : 'a -> 'a arbre -> 'a arbre * 'a arbre = <fun>
```


ajout à la racine (2/2)

```
# let ajout e a =
   let (g , d) = coupe e a in Noeud(e,g,d)
val ajout : 'a -> 'a arbre -> 'a arbre = <fun>
```



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 -> failwith "erreur, enleve_plus_grand"
  | 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 = <fun>
```


suppression d'un élément (2/2)

```
# 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)
val suppression : 'a -> 'a arbre -> 'a arbre = <fun>
```


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 $\log(n)$