Algorithmique et Approche Fonctionnelle de la Programmation

Cours 9 : Arbres de recherche

30 novembre 2009

Louis Mandel
Université Paris-Sud 11
louis.mandel@lri.fr

D'après des transparents de Sylvain Conchon

Recherche dans un arbre binaire

La fonction suivante permet de rechercher un élément dans un arbre binaire

```
# let rec recherche elt a =
    match a with
    | Vide -> false
    | Noeud(x,g,d) ->
        x = elt || recherche elt g || recherche elt d
  ;;
val recherche : 'a -> 'a arbre -> bool = <fun>
```

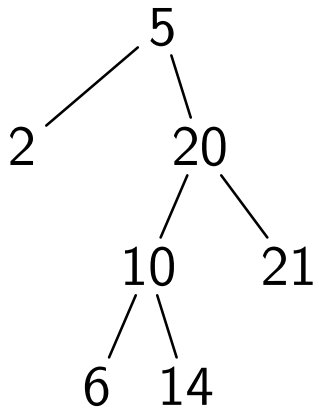
- Le temps de recherche est proportionnel à la taille de l'arbre $(O(n))$
- Il faut des hypothèses plus fortes sur la structure de l'arbre afin d'obtenir une meilleure complexité

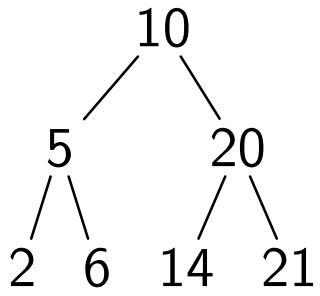
Arbres Binaires de Recherche

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 elt abr =
    match abr with
    | Vide -> false
    | Noeud (x,_,_) when x = elt -> true
    | Noeud (x,g,_) when elt < x -> recherche elt g
    | Noeud (_,_,d)  -> recherche elt d
  ;;
val recherche : 'a -> 'a arbre -> bool = <fun>
```

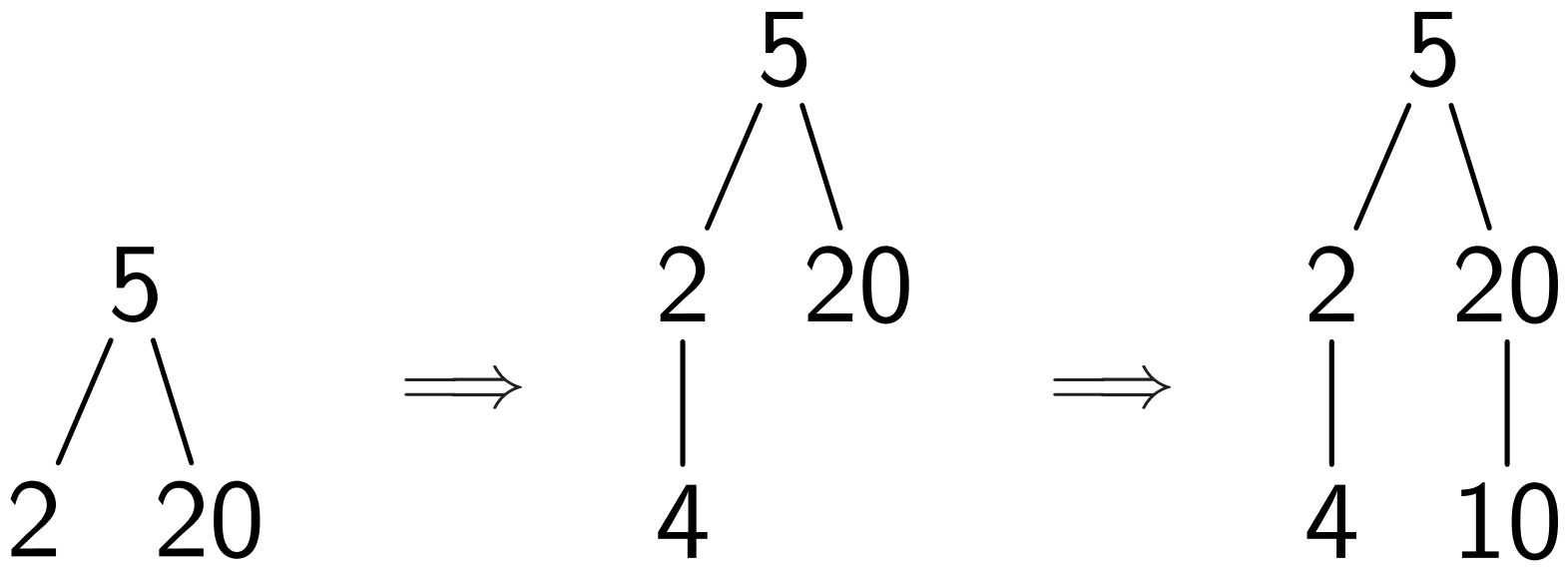
Ajout d'un élément

- 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 elt abr =
    match abr with
    | Vide -> Noeud(elt, Vide, Vide)
    | Noeud(x,_,_) when elt = x -> abr
    | Noeud(x,g,d) when x < elt -> Noeud(x, g, ajout elt d)
    | Noeud(x,g,d) -> Noeud(x, ajout elt g, d)
  ;;
val ajout : 'a -> 'a arbre -> 'a arbre = <fun>
```

Exemple d'ajout aux feuilles



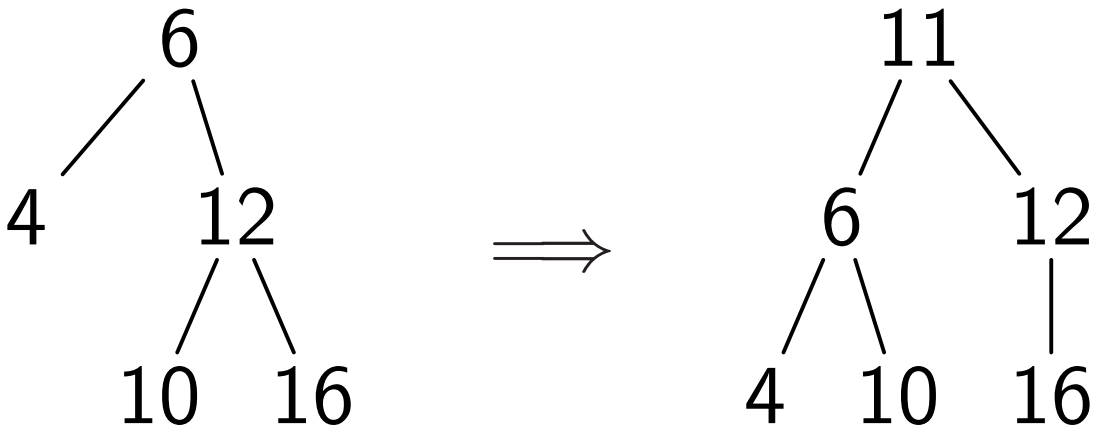
Ajout à la racine

- 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)`

```
# let rec coupe elt abr =
    match abr with
    | Vide -> (Vide , Vide)
    | Noeud(x,g,d) when x = elt -> (g , d)
    | Noeud(x,g,d) when x < elt ->
       let (t1, t2) = coupe elt d in
       (Noeud(x,g,t1), t2)
    | Noeud(x,g,d) ->
       let (t1, t2) = coupe elt g in
       (t1, Noeud(x,t2,d)) ;;
val coupe : 'a -> 'a arbre -> 'a arbre * 'a arbre = <fun>
```

Ajout à la racine

```
# 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

- 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 abr =
    match abr with
    | 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

```
# let rec suppression elt abr =
    match abr with
    | Vide -> Vide
    | Noeud(x, Vide, d) when x = elt -> d
    | Noeud(x, g, d) when x = elt ->
       let (y, g') = enleve_plus_grand g in Noeud(y, g', d)
    | Noeud(x, g, d) when elt < x ->
       Noeud(x, suppression elt g, d)
    | Noeud(x, g, d) ->
       Noeud(x, g, suppression elt 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 $\log(\mathrm{n})$