Cours numéro 9 : arbres binaires et de recherche

LI213 – Types et Structures de données

Christophe Gonzales – Pierre-Henri Wuillemin

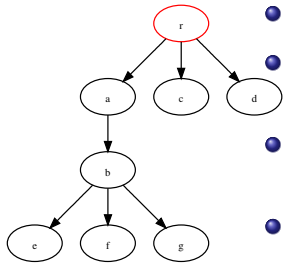
Licence d'Informatique – Université Paris 6

Arbre

Arbre

Un arbre est un ensemble fini A d'éléments, liés entre eux par une relation, dite de "parenté", vérifiant ces propriétés :

- Relation notée "x est le parent de y" ou "y est le fils de x"
- Il existe un unique élément r (**racine**) de A sans parent
- À part r, tout élément de A possède un **unique** parent



- Les éléments de A sont des **nœuds**
- Les nœuds sans fils sont des **feuilles** ou **nœuds terminaux**
- Les descendants d'un nœud x forment le **sous-arbre** de racine (ou issu de) x
- Un arbre **n-aire** est un arbre dont les nœuds ont tous au plus n fils.

Arbre binaire

Arbre binaire

- Un arbre binaire est un arbre 2-aire.
- Les fils sont spécialisés et nommés : fils **gauche** et fils **droit**.
  On confondra souvent le fils et le sous-arbre issu du fils.

Arbre binaire - définition récursive

Un arbre binaire A est défini par :

- L'arbre vide ($\emptyset$) est un arbre binaire
- l'arbre de racine r, de fils gauche A_g et de fils droit A_d est un arbre binaire si A_g et A_d sont des arbres binaires.

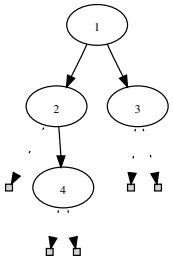
Représentation d'un arbre binaire par un type somme

Avec un enregistrement

```
type 'a cellule = { contenu : 'a;
                    fg : 'a arbre;
                    fd : 'a arbre }
 and 'a arbre = Nil | Noeud of 'a cellule
```

Avec un triplet (présentation infixe)

```
type 'a bintree = Nil |
          Node of 'a bintree * 'a * 'a bintree
```




```
# let arbre=Noeud{
    contenu=1;
    fg=Noeud{
      contenu=2;
      fg=Nil;
      fd=Noeud{contenu=4;
        fg=Nil;fd=Nil
      }
    };
    fd=Noeud{contenu=3;
      fg=Nil;fd=Nil}
  } ;;
```

```
# let tree=Node(
    Node(
      Nil,
      2,
      Node(Nil,4,Nil)
    )
    1,
    Node(Nil,3,Nil),
;; )
```

Fonctions simples (avec `bintree`)

- Récupérer le contenu du nœud racine

```
# let contenu a = match a with
    Nil -> failwith "contenu"
  | Node( _, x,_)-> x ;;
val contenu : 'a bintree -> 'a = <fun>
```

- Récupérer les fils (sous-arbres gauche et droit)

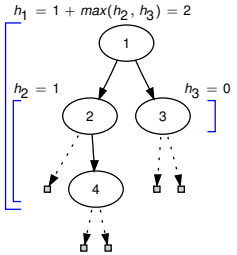
```
# let fg = function
    Nil -> failwith "fg"
  | Node(fg, _,_) -> fg;;
val fg : 'a bintree -> 'a bintree = <fun>

# let fd = function
    Nil -> failwith "fd"
  | Node( _, _,fd) -> fd;;
val fd : 'a bintree -> 'a bintree = <fun>
```


Fonctions simples (2)

- taille d'un arbre (nombre de nœuds)

```
# let rec taille a = match a with
    Nil -> 0
  | Node(fg,_,fd) ->1+(taille fg)+(taille fd) ;;
val taille : 'a bintree -> int = <fun>
```

- hauteur d'un arbre



$$h(A) = \begin{cases} -1 & \text{si } A = \emptyset \\ 1 + max(h(A_g), h(A_d)) & \text{sinon.} \end{cases}$$

```
# let rec hauteur = function
    Nil -> -1
  | Node(fg,_,fd) ->1 +
     max (hauteur fg) (hauteur fd) ;;
val hauteur : 'a bintree -> int = <fun>
```


Fonctions sur les arbres (3)

- fonction `elt_tree`

```
# let rec elt_tree x a = match a with
    Nil -> false
  | Node(fg,x,fd) -> true (*X dans motif !*)
  | Node(fg,y,fd) -> (x=y)
                  || (elt_tree x fg)
                  || (elt_tree x fd) ;;
val elt_tree : 'a -> 'a bintree -> bool = <fun>
```

- construction de `bintree` : fonction `map_tree`

```
# let rec map_tree f a = match a with
    Nil -> Nil
  | Node(fg,y,fd) -> Node(
                  map_tree f fg,
                  f y,
                  map_tree f fd ) ;;
val map_tree : ('a -> 'b) -> 'a bintree ->
          'b bintree = <fun>
```


Utilisation : représentation d'un dictionnaire

Un dictionnaire est un ensemble de couples `clé/valeur` où

- Les `clés` forment un ensemble **totalement ordonné**
- Chaque `valeur` est associée à une unique `clé`.
- Chaque `clé` caractérise une unique `valeur`.

Un dictionnaire sert à représenter un dictionnaire, une liste de contacts, une base de données triée par un identifiant, etc ...
Problème : comment représenter un dictionnaire ?

- un tableau trié ? On peut retrouver facilement une valeur (par dichotomie). Mais l'insertion et la suppression suppose de décaler (au pire) l'ensemble des éléments ...
- une liste triée ? L'insertion et la suppression sont rapides. La recherche est lente ...

Nouvelle proposition : Arbre Binaire de Recherche (ABR)

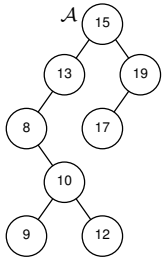
Définition

Dans ce qui suit, on utilisera $c(x)$ comme notation indiquant la clé du nœud x. Dans les programmes, on simplifiera le code en identifiant le contenu du nœud à la clé.

Arbre Binaire de Recherche

Un Arbre Binaire de Recherche est :

- un arbre binaire A
- $\forall x \in A, \forall y \in A_g(x), \forall z \in A_d(x), c(y) < c(x) < c(z)$

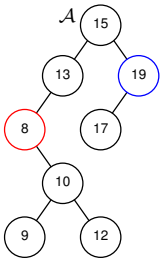


Le parcours **infixe** d'un ABR donne la liste triée des clés.

```
# let rec infixe a = match a with
    Nil -> []
  | Node(fg,c,fd) -> (infixe fg) @ [c] @ (infixe fd)
;;
val infixe : 'a bintree -> 'a list = <fun>
```

infixe $\mathcal{A}$ ::= [8;9;10;12;13;15;17;19]

Recherche du plus petit/grand élément



Dans un ABR :

- La plus petite clé se trouve dans le nœud le "plus à gauche"
  Décalage vers la droite = augmentation de la clé
- La plus grande clé se trouve dans le nœud le "plus à droite"
  Décalage vers la gauche = diminution de la clé

```
# let rec getMin = function
    Nil -> failwith "getMin"
  | Node(Nil,c,_) -> c
  | Node(fg,_,_) -> getMin fg
;;
```

```
# let rec getMax = function
    Nil -> failwith "getMax"
  | Node(_,c,Nil) -> c
  | Node(_,_,fd) -> getMax fd
;;
```


Vérification de la structure d'un ABR

Problème : Vérifier qu'un arbre binaire est un ABR.

```
# let rec checkABR = function
  Nil -> true
| Node(Nil, _,Nil) -> true
| Node(Nil,c,fd) -> (checkABR fd) && (c<getMin fd)
| Node(fg,c,Nil) -> (checkABR fg) && (getMax fg<c)
| Node(fg,c,fd) -> (checkABR fd) && (c<getMin fd)
                 && (checkABR fg) && (getMax fg<c)
;;
val checkABR : 'a bintree -> bool = <fun>
```

Remarque : La fonction $(<)$ pourrait être passée en argument pour paramétrer le choix de la relation d'ordre sur les clés.

Recherche d'un élément par sa clé

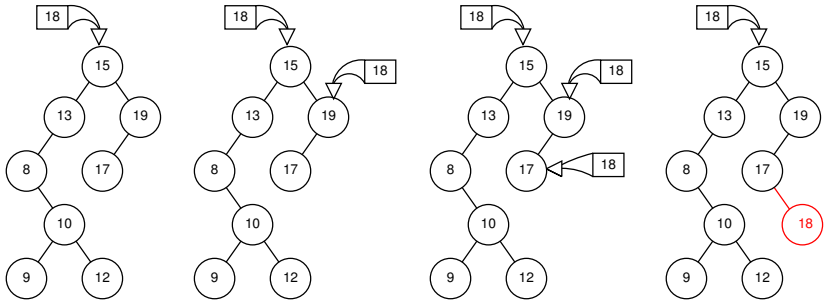
Problème : Vérifier qu'un élément (une clé) est dans un ABR.

```
# let rec eltABR x = function
    Nil -> false
  | Node(_,c,_) when c=x -> true
  | Node(fg,c,fd) -> if (x<c) then
                       eltABR x fg
                     else
                       eltABR x fd
;;
val eltABR : 'a -> 'a bintree -> bool = <fun>
```

Remarque : La fonction $(<)$ pourrait être passée en argument pour paramétrer le choix de la relation d'ordre sur les clés.

Insertion d'un élément

Problème : Insérer un nouvel élément (une nouvelle clé).



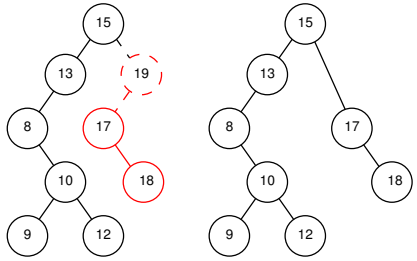

```
#  let insertion e = function
    Nil -> Node(Nil,e,Nil)
  | Node(fg,c,fd) ->
     if (e<c) then
        Node(insertion e fg,c,fd)
     else
        Node(fg,c,insertion e fd)
  ;;
```


Suppression d'un élément (1)

Sous-problème : Suppression du max d'un ABR

Remarques :

- la fonction va rendre un couple : max,ABR \ {max}
- propriété utilisée : **un max n'a pas de sous-arbre droit**




```
# let rec supprMax = function
    Nil -> failwith "supprMax"
  | Node(fg,m,Nil) -> (m,fg)
  | Node(fg,rac,fd) ->let (m,nvfd)=supprMax fd in
     (m ,Node(fg,rac,nvfd))
;;
val supmax : 'a bintree -> 'a * 'a bintree = <fun>
```


Suppression d'un élément (2)

Idées de l'algorithme :

- Si l'élément n'est pas la racine de l'arbre, il suffit de le supprimer dans le bon fils (gauche ou droit) et de recréer l'arbre complet à partir de ce nouveau fils.
- Si l'élément est la racine, alors le bon candidat à son remplacement est soit son fils s'il n'en a qu'un, soit le max de son fils gauche (ou le min de son fils droit)

```
# let rec suppABR e = function
    Node(fg,c,Nil) when c=e -> fg
  | Node(Nil,c,fd) when c=e -> fd
  | Node(fg,c,fd) when c=e -> let (m,neofg)=supmax fg in
                      Node(neofg,m,fd)
  | Node(fg,c,fd) when c<e -> Node(fg,c,suppABR e fd)
  | Node(fg,c,fd) when c>e -> Node(suppABR e fg,c,fd)
  | _ -> failwith "suppABR"
;;
val suppABR : 'a bintree -> 'a bintree = <fun>
```


Intérêt des ABR

- Les Arbres Binaires de Recherche sont une structure de données qui permet de représenter correctement un ensemble ordonné de clés.
- Les alternatives (tableaux ou listes) ont de bien moins bons comportements dans les opérations usuelles : ajout / suppression / recherche.
- tous les algorithmes des ABRs, le temps de calcul est proportionnel à la hauteur de l'arbre (et non au nombre de clés comme pour les autres alternatives)
- **toutefois**, dans le pire des cas, la hauteur de l'arbre est le nombre de clés dans l'arbre.
- En moyenne, les ABRs sont quand même souvent intéressants.
- Pour éviter les mauvais cas : **équilibrage des arbres**.