PLAN DU COURS 9

« Programmation récursive » - Saison 2

Structures arborescentes : arbres binaires de recherche

- Définition d'un arbre binaire de recherche
- Efficacité
- Recherche et ajout d'un élément
- Suppression d'un élément

ARBRE BINAIRE DE RECHERCHE

Définition : *Un arbre binaire de recherche est un arbre binaire tel que*

- *l'étiquette de la racine est*
  - *supérieure à toutes les étiquettes du sous-arbre gauche*
  - *inférieure à toutes les étiquettes du sous-arbre droit*
- *les sous-arbres gauche et droit de la racine sont aussi des arbres binaires de recherche*

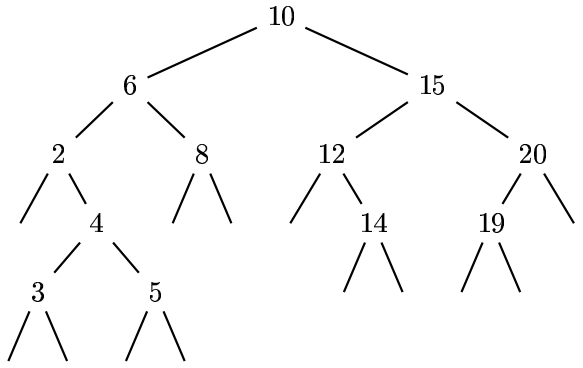
Propriété IC : *Si A est un arbre binaire de recherche, alors la liste* ***Infixe*** *des étiquettes de A est en ordre* ***Croissant***

Type : `ArbreBinRecherche` = `ArbreBinaire[Nombre]/IC vraie/`

LISTE INFIXE

```
;;; liste-infixe :  ArbreBinaire[α] -> LISTE[α]
;;; (liste-infixe B) rend la liste infixe de B
(define (liste-infixe B)
  (if (ab-noeud? B)
      (append (liste-infixe (ab-gauche B))
              (list (ab-etiquette B))
              (liste-infixe (ab-droit B)))
      (list)))
```

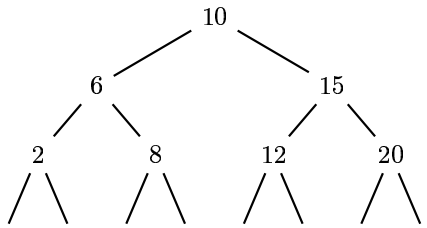

EXEMPLE



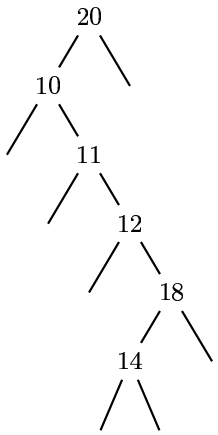
liste infixe : `2 3 4 5 6 8 10 12 14 15 19 20`

Des ABR extrêmes

« bien équilibré »



« dégénéré »



Structure de données

Représenter un ensemble d'éléments (ordre total), sur lequel on effectue les opérations suivantes :

- recherche d'un élément
- ajout d'un élément
- suppression d'un élément

Principe des algorithmes

Une *unique comparaison* avec l'étiquette de l'arbre permet d'aiguiller le traitement

- *soit* dans le sous-arbre gauche
- *soit* dans le sous-arbre droit

Le principe d'aiguillage permet de « laisser tomber » un sous-arbre entier (gauche ou droit) et de recommencer récursivement le traitement sur *un seul* sous-arbre.

Schéma de fonction

```
;;; f-abr: Nombre * ArbreBinRecherche -> ...
(define (f-abr x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette ABR)))
        (cond
          ((= x e) ...)
          ((< x e) (combinaison (f-abr x (ab-gauche ABR))))
          (else (combinaison (f-abr x (ab-droit ABR)))) ) )
      cas-arbre-vide ))
```

Efficacité

Si chaque sous-arbre contient à peu près la moitié des nœuds de l'arbre entier (arbre équilibré), alors le nombre de noeuds restant à examiner est *divisé par deux* à chaque fois : **dichotomie**

$$n \to n \div 2 \to n \div 4 \to \ldots \to n \div 2^p \Longrightarrow p \approx \log_2 n$$

Dichotomie => temps d'exécution logarithmique $0(\log n)$

MAIS si l'arbre est dégénéré le nombre de noeuds restant à examiner est seulement *diminué de 1* à chaque fois

$$n \to n-1 \to n-2 \to n-3 \ldots$$

=> temps d'exécution linéaire $0(n)$

Recherche d'un élément

La fonction de recherche est un semi-prédicat

```
;;; abr-recherche: Nombre * ArbreBinRecherche -> ArbreBinRecherche + #f
;;; (abr-recherche x ABR) rend l'arbre de racine x, lorsque x est dans ABR
;;; et renvoie #f si x n'apparaît pas dans ABR
(define (abr-recherche x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette  ABR)))
        (cond ((= x e) ABR)
              ((< x e) (abr-recherche x (ab-gauche ABR)))
              (else (abr-recherche x (ab-droit ABR))) ) )
      #f))
```


Ajout d'un élément

Principe : ajouter « à l'endroit où » la recherche s'est terminée en échec

Remarque : la fonction d'ajout rend comme résultat un arbre binaire de recherche qui est (re)construit au fur et à mesure (constructeur `ab-noeud`)

Fonction d'ajout

```
;;; abr-ajout: Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;; (abr-ajout x ABR) rend l'arbre ABR lorque x est dans ABR et sinon
;;; rend un arbre bin. de rech. qui contient x et toutes les étiquettes d'ABR
(define (abr-ajout x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette  ABR)))
        (cond ((= x e) ABR)
              ((< x e) (ab-noeud e
                                 (abr-ajout x (ab-gauche ABR))
                                 (ab-droit ABR)))
              (else (ab-noeud e
                              (ab-gauche ABR)
                              (abr-ajout x (ab-droit ABR))))))
      (ab-noeud x (ab-vide) (ab-vide))))
```

Suppression d'un élément

Principe :

- rechercher le sous-arbre dont l'élément à supprimer est la racine
- effectuer la suppression
  - la suppression est simple si l'élément n'est pas relié à 2 sous-arbres
  - la suppression est plus complexe sinon : modifier le sous-arbre en remplaçant la racine par son « précédent » dans le sous-arbre. *« précédent » ≡ le plus grand des éléments inférieurs à la racine*
    - les éléments inférieurs forment le sous-arbre gauche
    - le plus grand des inférieurs est au bout de la branche droite

Remarque : autre façon pour supprimer la racine d'un ABR : la remplacer par son « suivant » (le plus petit des éléments supérieurs à la racine)

Fonction de suppression

```
;;; abr-moins: Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;; (abr-moins x ABR) rend ABR lorque x n'y est pas et sinon rend un
;;; arbre bin. de rech. qui contient toutes les étiquettes de ABR sauf x
(define (abr-moins x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette ABR)))
        (cond ((= x e) (moins-racine ABR)) ; supprimer la racine
              ((< x e) (ab-noeud e
                                 (abr-moins x (ab-gauche ABR))
                                 (ab-droit ABR)))
              (else (ab-noeud e
                              (ab-gauche ABR)
                              (abr-moins x (ab-droit ABR))))))
      (ab-vide)))
```

Fonction de suppression de la racine

```
;;; moins-racine: ArbreBinRecherche/non vide/ -> ArbreBinRecherche
;;; (moins-racine ABR) rend l'arbre binaire de recherche qui contient toutes
;;; les étiquettes qui apparaissent dans ABR hormis l'étiquette de sa racine.
(define (moins-racine ABR)
  (cond ((not (ab-noeud? (ab-gauche ABR)))
         (ab-droit ABR))
        ((not (ab-noeud? (ab-droit ABR)))
         (ab-gauche ABR))
        (else ; défaire le ss-ab-g -> max + ABR privé du max
         ... )))
```


Le max d'un arbre binaire de recherche

Cette fonction renvoie le max d'un arbre binaire de recherche

```
;;; max-abr : ArbreBinRecherche -> Nombre
;;; (max-abr ABR) rend le max d'un arbre binaire de recherche
;;; HYPOTHÈSE : ABR est non vide
(define (max-abr ABR)
  (if (ab-noeud? (ab-droit ABR))
      (max-abr (ab-droit ABR))
      (ab-etiquette ABR)))
```


Arbre binaire de recherche sans son max

Cette fonction renvoie un ABR

```
;;; abr-sauf-max : ArbreBinRecherche -> ArbreBinRecherche
;;; HYPOTHÈSE : ABR est non vide
;;; (abr-sauf-max ABR) rend un arbre binaire de recherche qui contient
;;; toutes les étiquettes qui apparaissent dans ABR, hormis ce maximum.
(define (abr-sauf-max ABR)
  (if (ab-noeud? (ab-droit ABR))
      (ab-noeud (ab-etiquette ABR)
                (ab-gauche ABR)
                (abr-sauf-max (ab-droit ABR)))
      (ab-gauche ABR)))
```

Le max ET de l'arbre privé du max

```
;;; max-sauf-max : ArbreBinRecherche -> NUPLET[Nombre ArbreBinRecherche]
;;; HYPOTHÈSE : ABR est non vide
;;; (max-sauf-max ABR) rend le couple formé de la plus grande étiquette
;;; de l'arbre ABR et d'un arbre binaire de recherche qui contient toutes
;;; les étiquettes qui apparaissent dans ABR, hormis ce maximum.
(define (max-sauf-max ABR)
  (if (ab-noeud? (ab-droit ABR))
      (let ((m-sm-ss-ab-d (max-sauf-max (ab-droit ABR))))
        (list (car m-sm-ss-ab-d)
              (ab-noeud (ab-etiquette ABR)
                        (ab-gauche ABR)
                        (cadr m-sm-ss-ab-d))))
      (list (ab-etiquette ABR) (ab-gauche ABR))))
```


UPMC MIAS1-Info1 Année 2003-2004« Programmation récursive » Cours 9 Anne Brygoo, Michèle Soria

Fonction de suppression de la racine

```
;;; moins-racine : ArbreBinRecherche/non vide/ -> ArbreBinRecherche
;;; (moins-racine ABR) rend l'arbre binaire de recherche qui contient
;;; toutes les étiquettes qui apparaissent dans ABR hormis l'étiquette
;;; de sa racine.
(define (moins-racine ABR)
  (cond ((not (ab-noeud? (ab-gauche ABR)))
         (ab-droit ABR))
        ((not (ab-noeud? (ab-droit ABR)))
         (ab-gauche ABR))
        (else ; défaire le ss-ab-g ->  max + ABR privé du max
         (let ((m-sm-ss-ab-g (max-sauf-max (ab-gauche ABR))))
           (ab-noeud (car m-sm-ss-ab-g)
                     (cadr m-sm-ss-ab-g)
                     (ab-droit ABR))))))
```


UPMC MIAS1-Info1 Année 2003-2004« Programmation récursive » Cours 9 Anne Brygoo, Michèle Soria