DEUG MIAS 2^e^ année

TD de programmation fonctionnnelle n°3

correction

Les arbres (suite)

Pierre Boulet   Jean-Marc Nicod   Frédéric Vivien

Le but de ce TD-TP est d'utiliser la structure d'arbre pour trier une liste de valeurs. Pour cela, on utilise la construction d'un arbre de recherche. *Dans un souci de simplification et de généralisation, les arbres seront représenté par une liste. Le premier élément de cette liste est l'étiquette et la suite est la liste des fils. Un arbre binaire est donc un arbre donc chaque noeud est une liste à trois éléments.*

TD : Tri avec les arbres de recherche

Définition

On définit un arbre de recherche de la manière suivante : l'étiquette de tout nœud ou de toute feuille appartenant au sous arbre **gauche** d'un arbre de recherche est inférieure à la valeur de l'étiquette de cet arbre; au contraire, l'étiquette de tout nœud ou de toute feuille appartenant au sous arbre **droit** d'un arbre de recherche est supérieure ou égale à la valeur de l'étiquette de cet arbre. Cette définition s'applique à tous les sous-arbres d'un arbre de recherche. La figure 1 est un exemple d'arbre de recherche.

1 Construction d'un arbre de recherche

Question 1.1 : Les fonctions *make-arbre*, *make-feuille*, *etiquette*, *fg*, *fd* et *hauteur* sont données ici pour la représentation que nous avons adoptée pour les arbres binaires.

```
(define make-arbre
  (lambda (etiq fg fd)
    (list etiq fg fd)))
```

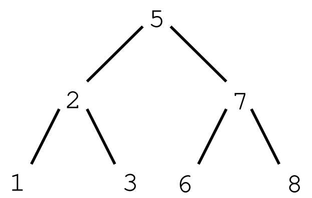

Fig. 1 - *Définition d'un* arbre de recherche

```
(define make-feuille
  (lambda (etiq)
    (list etiq '() '())))

(define etiquette
  (lambda (a)
    (if (null? a)
        '()
        (car a))))

(define fg
  (lambda (a)
    (if (null? a)
        '()
        (cadr a))))

(define fd
  (lambda (a)
    (if (null? a)
        '()
        (caddr a))))

(define hauteur
  (lambda (a)
    (if (null? a)
        0
        (+ 1 (max (hauteur (fg a)) (hauteur (fd a)))))))
```

Question 1.2 : Ecrire la fonction *insere* qui permet l'insertion d'un élément dans un arbre de recherche.

```
(define insere
  (lambda (a e)
    (if (null? a)
        (make-feuille e)
        (if (> (etiquette a) e)
            (make-arbre (etiquette a)
                        (insere (fg a) e) (fd a))
            (make-arbre (etiquette a)
                        (fg a) (insere (fd a) e))))))
```

Question 1.3 : Ecrire la fonction *make-arbre-rech* qui permet la création d'un arbre de recherche à partir d'une liste de valeurs. Etant donné que le premier élément de la liste s'insère dans un arbre de recherche vide, la récurcivité peut être placée soit dans une autre fonction, soit dans la fonction *make-arbre-rech*, dans un **letrec**.

```
(define make-arbre-rech
  (lambda (l)
    (if (null? l)
        '()
        (letrec
          ((arbre-rech (lambda (a l)
            (if (null? l)
                a
                (arbre-rech (insere a (car l)) (cdr l))))))
        (arbre-rech '() l)))))
```

2 Utilisation des arbres de recherche

Question 2.1 : Ecrire la fonction *in-arbre-rech?* qui permet de chercher un élément dans un arbre de recherche. Il faut exploiter la manière dont sont organisées les informations dans un tel arbre pour que cette recherche soit la plus rapide.

```
(define in-arbre-rech?
  (lambda (a e)
    (if (null? a)
        #f
        (if (eqv? (etiquette a) e)
            #t
            (if (> (etiquette a) e)
                (in-arbre-rech? (fg a) e)
                (in-arbre-rech? (fd a) e))))))
```

Question 2.2 : Ecrire une fonction *tri-liste* qui permet de trier une liste de valeurs dans l'ordre croissant. Pour cela, utilisez les fonctions qui ont été écrites précédemment.

```
(define parcours-prof
  (lambda (a)
    (if (null? a)
        '()
        (append (parcours-prof (fg a))
                (list (etiquette a))
                (parcours-prof (fd a))))))

(define tri-liste
  (lambda (l)
    (if (null? l)
        '()
        (parcours-prof (make-arbre-rech l)))))
```

3 Equilibrage des **arbres de recherche**

Suivant l'ordre dans lequel les valeurs de la liste à trier sont placées, les **arbres de recherche** peuvent être très différents. Le cas le plus flagrant est la construction d'un **arbre de recherche** à partir d'une liste déjà triée. Dans ce cas, l'**arbre de recherche** ressemble plus à une liste chaînée qu'à un arbre binaire. Or plus une valeur doit être insérée profondément dans l'arbre, plus cette insertion est longue. Il est donc intéressant de pouvoir équilibrer des **arbre de recherche** ou de savoir les construire directement équilibrés.

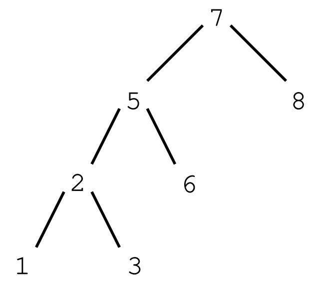

FIG. 2 - *Exemple d'un* arbre de recherche *non équilibré*

La figure 2 illustre le cas d'un **arbre de recherche** non équilibré. Nous notons un déséquilibre dans un arbre dès que la différence entre les hauteurs des deux branches d'un sous-arbre est supérieure ou égale à 2. Un algorithme de rééquilibrage pourrait être d'utiliser des transformations locales appelées ***rotations*** : il s'agit de faire basculer vers la droite (resp. gauche) un sous-arbre qui est trop déséquilibré vers la gauche (resp. droite). Il y a 4 type de rotation : les *rotations simples*, à droite et à gauche, et les *rotations doubles*, rotation gauche-droite et rotation droite-gauche. La figure 3 illustre une *rotation droite*. On doit prendre en compte **5 cas** différents pour l'équilibrage d'un arbre de recherche **a** :

- **cas** 1. l'arbre **a** est équilibré, donc il n'y a rien à faire.
- **cas** 2. la différence de hauteur entre le fils gauche est le fils droit est de 2.
  - **cas** 2.1. cette différence est portée par la branche gauche du fils gauche de **a** $\Rightarrow$ *rotation gauche* au niveau de la racine de **a** (figure 3).
  - **cas** 2.2. cette différence est portée par la branche droite du fils gauche de **a** $\Rightarrow$ *double rotation gauche-droite* Il s'agit en fait d'une *rotation droite* au niveau du fils gauche de **a** puis d'une *rotation gauche* au niveau de la racine de **a** (figure 4).
- **cas** 3. la différence de hauteur entre le fils droite est le fils gauche est de 2.
  - **cas** 3.1. c'est le cas symétrique du cas 2.1.
  - **cas** 3.2. c'est le cas symétrique du cas 2.2.

Question 3.1 : Ecrire les fonctions *rotation-g* et *rotation-d* qui permettent de rétablir l'équilibre entre les deux branches d'un sous arbre déséquilibré.

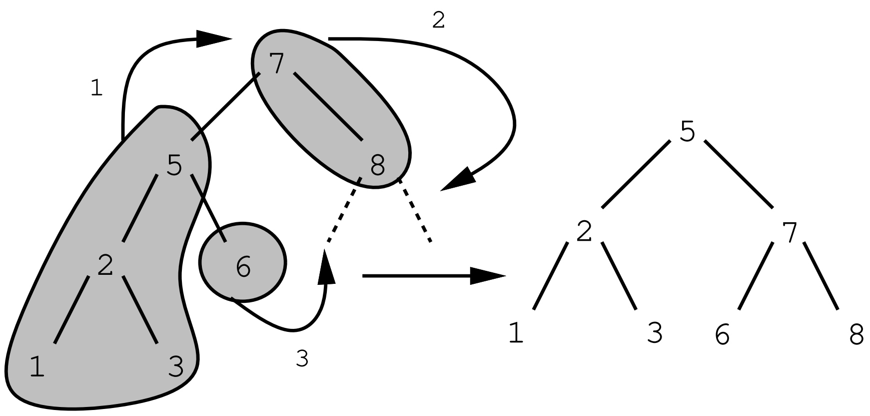

FIG. 3 - *rotation droite*

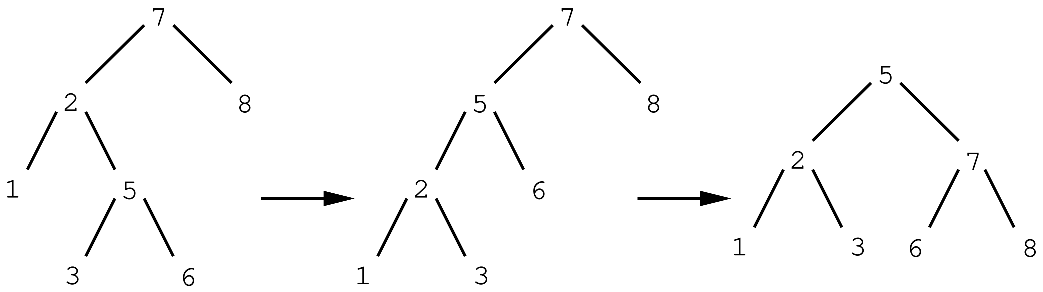

FIG. 4 - *rotation gauche-droite*

```
(define rotation-g
  (lambda (a)
    (make-arbre (etiquette (fd a))
                (make-arbre (etiquette a)
                            (fg a)
                            (fg (fd a)))
                (fd (fd a)))))

(define rotation-d
  (lambda (a)
    (make-arbre (etiquette (fg a))
                (fg (fg a))
                (make-arbre (etiquette a)
                            (fd (fg a))
                            (fd a)))))
```

Question 3.2 : Ecrire la fonction *equilibre-arbre-rech* qui permet d'équilibrer un arbre de recherche créé par la fonction *make-arbre-rech* à partir d'une liste de valeur.

```
(define equilibre-arbre-rech
  (lambda (a)
    (if (<= (hauteur a) 1)
        a
        (begin
          (define fils-g (equilibre-arbre-rech (fg a)))
          (define fils-d (equilibre-arbre-rech (fd a)))
          (define arbre (make-arbre (etiquette a) fils-g fils_d))
          (cond
            ((>= 1 (abs (- (hauteur fils-g) (hauteur fils-d))))
             a)
            ((< 1 (- (hauteur fils-g) (hauteur fils-d)))
             (if (< 0 (- (hauteur (fd fils-g)) (hauteur (fg fils-g))))
                 (equilibre-arbre-rech
                   (rotation-d (make-arbre
                                 (etiquette arbre)
                                 (rotation-g fils-g)
                                 fils-d)))
                 (equilibre-arbre-rech (rotation-d arbre))))
            (else
              (if (< 0 (- (hauteur (fg fils-d)) (hauteur (fd fils-d))))
                  (equilibre-arbre-rech
                    (rotation-g (make-arbre
                                  (etiquette arbre)
                                  fils-g
                                  (rotation-d fils-d))))
                  (equilibre-arbre-rech (rotation-g arbre)))))))))
```

Question 3.3 : Ecrire la fonction *insere-eq* qui permet d'insérer une valeur dans un arbre de recherche équilibré; l'arbre de recherche résultat est équilibré. C'est la fonction *insere-eq* qui sera utilisée à l'avenir dans la fonction *make-arbre-rech*. Les rotations se font en remontant de la feuille d'insertion jusqu'à la racine, si nécessaire. Il faut utiliser la fonction *eq-ssarbre* dans la fonction *insere-eq*. La fonction *eq-ssarbre* est la version non récurcive de *equilibre-arbre-rech*.

```
(define eq-ssarbre
  (lambda (a)
    (if (< (hauteur a) 2)
        a
        (cond
         ((>= 1 (abs (- (hauteur (fg a)) (hauteur (fd a)))))
          a)
         ((< 1 (- (hauteur (fg a)) (hauteur (fd a))))
          (if (< 0 (- (hauteur (fd (fg a))) (hauteur (fg (fg a)))))
              (rotation-d (make-arbre
                           (etiquette a)
                           (rotation-g (fg a))
                           (fd a)))
              (rotation-d a)))
         (else
          (if (< 0 (- (hauteur (fg (fd a))) (hauteur (fd (fd a)))))
              (rotation-g (make-arbre
                           (etiquette a)
                           (fg a)
                           (rotation-d (fd a))))
              (rotation-g a)))))))

(define insere-eq
  (lambda (a e)
    (if (null? a)
        (make-feuille e)
        (if (> (etiquette a) e)
            (eq-ssarbre (make-arbre (etiquette a)
                                    (insere-eq (fg a) e)
                                    (fd a)))
            (eq-ssarbre (make-arbre (etiquette a)
                                    (fg a)
                                    (insere-eq (fd a) e)))))))
```

Question 3.4 : Ecrire une fonction qui donne comme résultat la liste triée de la liste de valeur donnée en paramètre.