Parcours d'arbre

Jean-Baptiste Rouquier

1er janvier 2005

1 Arbres

On représentera des arbres binaires avec information sur les nœuds internes et sur les feuilles par le type suivant :

```
type ('a,'b) arbre =
  | Noeud of 'a * ('a,'b) arbre * ('a,'b) arbre
  | Feuille of 'b
```

Toutes les fonctions devront être testées sur un exemple raisonnable.

Definition. Un ***tas*** (en anglais *heap*) est un arbre tel que l'étiquette de chaque nœud est plus petite, au sens large, que celle de chacun de ses fils.

Les types des nœuds et des feuilles sont donc identiques : un tas est de type `('a, 'a) arbre`.

Question 1.1. Où est le plus petit élément ? Le plus grand ?

Question 1.2. Écrire la fonction `test_racine : (int,int) arbre -> bool` qui vérifie que cette propriété (être plus petit que chacun de ses deux fils) est vérifiée pour la racine.

2 Parcours en profondeur

Le schéma général d'un parcours en profondeur d'abord est présenté sur l'algorithme 1. Le premier appel se fait sur la racine, et déclenche les appels récursifs sur tous les nœuds.

Algorithme : parcours_profondeur(x)

1 traiter le sommet x
if x n'est pas une feuille **then**
2 | parcours_profondeur(fils gauche de x)
3 | parcours_profondeur(fils droit de x)

Algorithme 1: parcours en profondeur

Question 2.1. Écrire une fonction `is_heap : int int arbre -> bool` qui parcours son argument en profondeur et vérifie s'il est bien un tas.

Question 2.2. L'algorithme présenté ici traite un sommet (ligne 1) avant de parcourir ses fils (lignes 2 et 3). C'est un parcours ***préfixe***. Il est aussi possible de traiter le sommet après avoir parcouru les fils, on parle alors de parcours ***postfixe***, voire après avoir parcouru le fils gauche et avant de parcourir le fils droit, ce qui constitue un parcours ***infixe***.

Implémenter la fonction `parcours_infixe : (int,int) arbre -> unit` qui parcours l'arbre dans l'ordre infixe et pour chaque nœud imprime le nom de son constructeur («nœud interne» ou «feuille») suivi de son étiquette.

Question 2.3 (bonus). On pose `type noeud = Node of int | Leaf of int` pour représenter un noeud isolé (sans ses fils) d'un arbre dont les étiquettes sont des entiers.

Écrire la fonction `parcours_postfixe (int,int) arbre -> noeud list` qui effectue un parcours postfixe, et qui renvoie la liste des noeuds visités, dans l'ordre où ils ont été visités.

Question 2.4 (bonus). Écrire la fonction `reconstruit : noeud list -> (int,int) arbre`, réciproque de `parcours_postfixe`.

Question 2.5 (bonus). Est-il possible d'écrire une fonction équivalente à `reconstruit` pour le parcours préfixe ? Pour le parcours infixe ?

3 Parcours en largeur

Le parcours en profondeur commence donc par descendre au plus profond d'un sous-arbre avant d'explorer une autre branche. L'idée du parcours en largeur est au contraire de traiter tous les les nœuds d'une profondeur donnée avant de passer aux nœuds plus profonds.

Le schéma général d'un parcours en largeur d'abord est donné par l'algorithme 2. Il utilise une file (FIFO) à_traiter.

Algorithme : parcours_largeur

(*initialisation*)
à_traiter := {racine}

(*boucle*)
while à_traiter $\neq \varnothing$ **do**
  x := take(à_traiter)
  traiter le sommet x
  if x n'est pas une feuille **then**
    ajouter le fils gauche de x à à_traiter
    ajouter le fils droit de x à à_traiter

Algorithme 2: parcours en largeur

Question 3.1. Montrer que cet algorithme termine.

Question 3.2. Implémenter la fonction générale

```
parcours_largeur : ('a -> unit) -> ('b -> unit) -> ('a,'b) arbre -> unit
```

Ainsi l'invocation `parcours_largeur f g t` parcours l'arbre `t` en largeur d'abord, en appelant la fonction `f` sur les nœuds internes et `g` sur les feuilles.