Option Informatique
Exercices sur les arbres

Sujet

4 avril 2007

1 Arbres peignes

Question 1

a *Donner des exemples d'arbre peigne.*

b *Montrer que l'on a l'équivalence : T est un arbre peigne ssi.* hauteur(T) = taille(T)*, où la* taille *est le nombre de noeuds internes de T.*

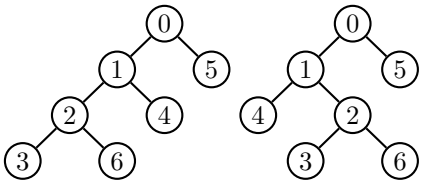
c *que devient cette relation avec le nombre de feuilles de T puis avec le nombre total de noeuds de T ?*

d *On définit le type des arbres par :*

```
type ('n,'f) arbre =
  feuille of 'f
  | noeud of ('n,'f) arbre * 'n * ('n,'f) arbre ; ;
```

Écrire une fonction CAML ***est_peigne*** *de type* ***('n,'f) arbre → bool***

a



b On sait déjà que hauteur(T) ≤ taille(T). Il suffit de montrer l'autre inclusion. Par induction sur la structure d'arbre peigne :
- Si l'arbre est une feuille, alors taille et hauteurs sont nulles, et égales.
- Si l'arbre est de la forme ($T = (a, r, T')$), alors par IH, hauteur(T') = taille T'. hauteur(T) = $1 + max$(hauteur(a) + hauteur(T')) = 1 + hauteur(T') = 1 + taille(T') = taille(T)
- Si ($T = (T', r, a)$), on obtient le même résultat qu'au cas précédent par symétrie.

c
```
let rec est_peigne = fun
| (feuille _) → true
| (noeud (feuille _,_,noeud a))→
                    (est_peigne (noeud a))
| (noeud (a,_,feuille _))→ (est_peigne a)
| _ → false ; ;
```

2 Recherche du minimum et du maximum d'un arbre T

Question 2 *Donner une version* CAML *de* ***min_max****. On précisera le type arbre utilisé.*

La difficulté réside dans le fait de fonder la récursion : on ne peut pas utiliser une instruction `failwith`, sinon l'appel récursif de la fonction allant jusqu'au feuilles, on échouera sur tout arbre. Il faut ici donner une valeur qui sera toujours perdante dans le «tournoi» de valeur qu'on renvoie.

```
let MaxE = 300 and MinE = (-300) ; ;

type arbre = nil | n of arbre * int * arbre ; ;

let rec min_max = function
  | nil → MaxE,MinE
  | n (g,x,d) → let (gmin,gmax) = (min_max g) and
        (dmin,dmax) = (min_max d) in
    (min (min gmin dmin) x),(max (max gmax dmax) x) ; ;
```

3 Parcours en largeur d'abord

Question 3

a *Que doit valoir* `L` *initialement ?*

b *On suppose que* `L` *contient la liste des arbres restant à traiter. On isole la tête de* `L` *: c'est l'arbre qui doit être traité. On le note (G, x, D). Que doit-on faire pour le traitement, la réactualisation de* `L`*, et la poursuite du parcours en largeur d'abord ? Proposer une structure plus adéquate pour* `L` *qu'une liste.*

c *Quand arrête-t-on le traitement ?*

d *En déduire une fonction récursive* ***traite_L fonc_f fonc_n*** *qui effectue le traitement en largeur d'abord des arbres de* `L`*.*

e *En déduire une fonction* ***largeur fonc_f fonc_n*** *qui effectue le traitement en largeur d'abord de* `T`*.*

f *En déduire une instruction* CAML *d'affichage des étiquettes des noeuds de l'arbre en largeur d'abord. On supposera $\mathcal{N} = \mathbb{R}$, et $\mathcal{F} = \mathbb{N}$.*

a `L` a initialement pour seul élément l'arbre dont on veut faire le parcours.

b On traite x, puis on concatène `[G ;D]` à la fin de la liste `L`, avant d'appeller récursivement le parcours sur la liste ainsi formée. Une structure qui économiserait le coût de la concaténation serait donc une pile (FIFO).

c
```
let rec traite_l fonc_f fonc_n = function
  |  [] → () ;
  | (feuille f) : :q → (fonc_f f)
  | (noeud (g,x,d)) : :q → begin
      fonc_n x ;
      traite_l fonc_f fonc_n (q©[g ;d])
    end ; ;
```

d
```
let largeur fonc_f fonc_n a =
  traite_l fonc_f fonc_n [a] ; ;
```

e
```
let affiche =
  largeur print_int print_float ; ;
```

4 Arbres et expressions algébriques

Question 4 *a Définir les types* `liste_EAP` *et* `arbre_EAP` *associés à L et $T(L)$.*

b Écrire une fonction `prefixe L` *qui renvoie un couple L, L' où L' est le plus long préfixe strict de L de poids égal à 1 (avec les poids vus en cours), et L'' est le reste de la liste L (i.e. $L = L'L''$).*

c Écrire une fonciton `liste2arbre L` *qui renvoie $T(L)$. Cette fonction s'arrêtera si L ne représente pas une EAP.*

d Écrire une fonction `arbre2listepre T` *qui renvoie la liste de l'EA préfixée associée à T.*

e Écrire une fonction `listpost2listpre L` *qui convertit une expression postfixée en expression préfixée.*

a
```
type EAP= nb of int| op of char
  and liste_EAP==EAP list
  and arbre_EAP= feuille of int
               | noeud of arbre_EAP*char*arbre_EAP ; ;
```

b On se sert d'une fonction auxiliaire `prefixe_aux` qui renvoie le plus long préfixe de poids 1 éventuellement non strict, puis on l'appelle sur la liste otée de son dernier élément, avant de corriger ce qu'elle renvoie dans le résultat final.

```
let rec poids=function
    |[] → 0
    |(nb(_)) : :q → poids q + 1
    |t : :q → poids q  - 1 ; ;

let prefixe l =
  let rec prefixe_aux candidat courant reste = function
    | [] → (rev candidat, reste)
    | t : :q →
        if (poids (t : :courant)) = 1 then
          prefixe_aux (t : :courant) (t : :cou-
rant) q q
        else
          prefixe_aux candidat (t : :courant) reste q
  in
  let a,b = (prefixe_aux [] [] [] (rev (tl (rev l)))) in
  a,b©[hd (rev l)] ; ;
```

c
```
let rec listeEAP2arbre l =
  let a,b = (prefixe l) in
  let c,d = (prefixe b) in
  if (a = [] or c = []) then
    match d with
      | [nb i] → feuille i
      | _ → failwith "non EAP"
  else
    match d with
      | [op p] →
        noeud ((listeEAP2arbre a),p,(listeEAP2arbre c))
      | _ → failwith "non EAP" ; ;
```

d
```
let rec arbre2liste = function
  | feuille f → [nb f]
  | noeud (g,x,d) →
    (arbre2liste g)©(arbre2liste d)©[op x] ; ;
```

e
```
let listepost2listpre l =
  let arbre2listepre = function
    | feuille f → [nb f]
    | noeud (g,x,d) →
      [op x]©(arbre2liste g)©(arbre2liste d)
  in
  (arbre2listepre (listeEAP2arbre l)) ; ;
```

5 Arbres équilibrés

Question 5 *Rappeller la fonction* `hauteur` *qui calcule la hauteur d'un arbre. on précisera le type arbre utilisé.*

```
type arbre = f of int | n of arbre * int * arbre ; ;

let rec hauteur = function
  | f i → 0
  | n (g,x,d) → 1+(max (hauteur g) (hauteur d)) ; ;
```

Question 6 *La profondeur d'un arbre est la longueur (en nombre d'arêtes) du plus court chemin de la racine à une feuille. Implémenter une fonction* `profondeur` *qui renvoie ce résultat.*

```
let rec profondeur = function
  | f i → 0
  | n (g,x,d) → 1+ (min (profondeur d) (profondeur d)) ; ;
```

Question 7 *Implémenter une fonction* `est_quasicomplet` *qui vérifie si un arbre est quasi-complet.*

```
let quasicomplet a =
  ((hauteur a) - (profondeur a)) <= 1 ; ;
```

Question 8 *Implémenter une fonction qui vérifie si un arbre binaire est un arbre H-équilibré. Déterminer la complexité de votre algorithme, vous paraît-elle raisonnable ?*

```
let rec h_equilibre = function
  | f _ - > true
  | n (g,i,d) as a - >
     ((hauteur g)-(hauteur d) <= 1) &
     (h_equilibre g) & (h_equlibre d)
```

Soit h la hauteur de l'arbre considéré. On se place dans le pire cas, qui est celui d'un arbre complet.

On montre aisément que la fonction `hauteur` ci-dessus effectue $2^{h_0} - 1$ opérations pour un arbre de hauteur h_0.

On effectue ici deux calculs de hauteur $h_0 - 1$, soit un calcul de hauteur h_0, pour chaque noeud racine d'un sous-arbre de hauteur h_0. On sent une borne grossière s'esquisser, mais tentons d'être plus précis.

On a 2^{h-h_0} noeuds racine d'un sous-arbre de hauteur h_0. Donc on effectue

$$\Sigma_{h_0=1}^{h} 2^{h-h_0} * (2^{h_0} - 1) = (h-1)2^h + 1 = O(h2^h)$$

La complexité de cette fonction est finalement «assez raisonnable» (comparée à un calcul de hauteur, par exemple).

6 Rotations sur un arbre binaire

Question 9

a Définir ces deux opérations en CAML*, évaluer leur complexité.*

b On définit la double rotation gauche-droite *par :*

$$GD[(T_g, x, T_d)] = D[G(T_g), x, T_d]$$

et la double rotation droite-gauche *par :*

$$DG[(T_g, x, T_d)] = G[T_g, x, D(T_d)]$$

Illustrer sur un dessin ce que font ces opérations. Donner leur complexité.

*c Montrer que ces opérations conservent la structure d'*arbre binaire de recherche.

d À l'aide de ces opérations, expliquer comment effectuer la supresison dans un arbre binaire de recherche.

Voir les *Éléments d'Algorithmique*, de Jean Berstel, Chapitre 6, page 6.

`http://www-igm.univ-mlv.fr/~berstel/Elements/EAChap6.pdf`