X 2003, option MP

Composition d'informatique

corrigé de M. Quercia (michel.quercia@prepas.org)

Remarque : le code Pascal donné pour la fonction `cons` est incorrect, la déclaration de la variable `res` doit être placée avant le corps de la fonction.

Question 1.
Dans le tableau ci-dessous, on a remplacé les 0 par des points pour une meilleure lisibilité.

	0	1	2	3	4	5	6	7
0	.	.	.	.	.	.	.	.
1	.	1	4	.	4	4	.	.
2	.	4	2	.	4	2	.	.
3	.	.	.	3	.	.	.	.
4	.	4	4	.	4	4	.	.
5	.	4	2	.	4	5	.	.
6	.	.	.	.	.	.	6	6
7	.	.	.	.	.	.	6	7

Question 2.
Il suffit de traduire la définition récursive de la hauteur donnée dans l'énoncé.

En Caml :
```
let rec hauteur pere a =
  if a = 0 then 0 else 1 + (hauteur pere pere.(a))
;;
```

En Pascal :
```
function hauteur(pere : vint; a:integer) : integer;
var h : integer;
begin
   h := 0;
   while a <> 0 do begin a := pere[a]; h := h+1 end;
   hauteur := h;
end;
```

Question 3.
On parcourt chaque liste du tableau `fils` donné en argument, et on inscrit `pere`$[i] = a$ pour chaque couple (i, a) tel que $a \in$ `fils`$.(i)$.

En Caml :
```
let filsEnPere fils =
  let n = vect_length(fils) in
  let pere = make_vect n 0  in
  let rec remplit i liste = match liste with
    | [] -> ()
    | a::suite -> pere.(a) <- i; remplit i suite
  in
  for i=0 to n-1 do remplit i fils.(i) done;
  pere
;;
```

En Pascal :

```
procedure filsEnPere(fils : vlint; var pere:vint);
var i : integer;
    f : lint;
begin
   for i:=0 to n-1 do begin
      f := fils[i];
      while f <> nil do begin pere[f^.val] := i; f := f^.suiv; end
   end
end;
```

Question 4.

Soient a, b deux nœuds de même hauteur h et c leur plus proche ancêtre commun de hauteur $h' \leqslant h$. Si $h' = h$ alors $c = a = b$. Sinon c est aussi ancêtre commun des pères de a et b, et c'est clairement le plus proche ancêtre commun de ces pères qui par ailleurs ont même hauteur. On en déduit une définition récursive de ppacMemeH :

$$\text{ppacMemeH}(a,b) = \begin{cases} a & \text{si } a = b \ ; \\ \text{ppacMemeH}(\text{pere}[a], \text{pere}[b]) & \text{sinon.} \end{cases}$$

Cette définition est bien fondée par récurrence sur la hauteur h commune à a et b.

Soient à présent a, b deux nœuds de hauteurs h_a, h_b et $c = \text{ppac}(a,b)$ de hauteur h. Si $h_a = h_b$ alors $c = \text{ppacMemeH}(a,b)$. Sinon, si par exemple $h_a > h_b$ alors $h < h_a$ donc c est aussi ancêtre du père de a, et c'est clairement le plus proche ancêtre commun à ce père et à b. On en déduit une définition récursive de ppac :

$$\text{ppac}(a,b) = \begin{cases} \text{ppacMemeH}(a,b) & \text{si } h(a) = h(b) \ ; \\ \text{ppac}(\text{pere}(a), b) & \text{si } h(a) > h(b) \ ; \\ \text{ppac}(a, \text{pere}(b)) & \text{si } h(a) < h(b). \end{cases}$$

A nouveau, cette définition est bien fondée par récurrence sur $\max(h_a, h_b)$.

En Caml :

```
let rec ppacMemeH pere a b =
  if a = b then a else ppacMemeH pere pere.(a) pere.(b)
;;
let ppac pere a b =

  (* x et y d\'esignent des noeuds de hauteurs hx et hy *)
  (* retourne ppac(x,y)                                 *)
  let rec calcule x hx y hy =
        if hx = hy then ppacMemeH pere x y
    else if hx > hy then calcule pere.(x) (hx-1) y hy
                    else calcule x hx pere.(y) (hy-1)

  in calcule a (hauteur pere a) b (hauteur pere b)
;;
```

En Pascal :

```
function ppacMemeH(pere : vint; a,b : integer) : integer;
begin
   while a <> b do begin a := pere[a]; b := pere[b] end;
   ppacMemeH := a;
end;

function ppac(pere : vint; a,b : integer) : integer;
var ha,hb : integer;
begin
   ha := hauteur(pere,a);
   hb := hauteur(pere,b);
```

```
   while ha > hb do begin a := pere[a]; ha := ha-1 end;
   while hb > ha do begin b := pere[b]; hb := hb-1 end;
   ppac := ppacMemeH(pere,a,b);
end;
```

Question 5.

On a $s_0 = 1$ et $s_k = 2s_{k-1}$ d'où $s_k = 2^k$ par récurrence sur k, et $n = 2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$.

Question 6.

Remarque : l'existence et l'unicité de la fonction ℓ semblent implicitement admises par l'énoncé, En ce qui concerne l'existence, on peut remarquer que si l'on effectue un parcours en ordre infixe de l'arbre B, alors la fonction ℓ qui à un nœud associe son numéro de visite satisfait aux conditions imposées. On en déduit *une* réponse possible pour l'arbre donné en exemple :

a	0	1	2	3	4	5	6
$\ell(a)$	4	5	2	1	7	6	3

Prouvons l'unicité de ℓ. Soient donc deux fonctions ℓ, ℓ' de B dans $\{1, \ldots, n\}$ telles que pour tout nœud interne a de fils gauche b et de fils droit c on ait :

$$\max_{u \in B_b} \ell(u) < \ell(a) < \min_{v \in B_c} \ell(v) \qquad \text{et} \qquad \max_{u \in B_b} \ell'(u) < \ell'(a) < \min_{v \in B_c} \ell'(v).$$

Dans un premier temps, on montre que ℓ est injective : soient deux nœuds a, b distincts et $c = \text{ppac}(a, b)$. Puisque $a \neq b$, on a $a \neq c$ ou $b \neq c$, par exemple $a \neq c$. c est donc un nœud interne et a appartient à l'une des deux branches issues de c. Si a appartient à la branche gauche issue de c alors b n'appartient pas à cette branche par définition de c donc $b = c$ ou b appartient à la branche droite issue de c. Dans les deux cas, $\ell(a) < \ell(c) \leqslant \ell(b)$, en particulier $\ell(a) \neq \ell(b)$. On conclut de même si a appartient à la branche gauche issue de c.

L'injectivité de ℓ est ainsi prouvée, et elle implique la bijectivité de ℓ puisque B a n nœuds. Par symétrie, ℓ' est aussi bijective. En particulier :

$$\sum_{a \in B} \ell(a) = \sum_{i=1}^{n} i = \sum_{a \in B} \ell'(a).$$

Considérons enfin $\ell'' = x \longmapsto \max(\ell(x), \ell'(x))$. On a encore la relation :

$$\max_{u \in B_b} \ell''(u) < \ell''(a) < \min_{v \in B_c} \ell''(v)$$

pour tout nœud a de fils gauche b et de fils droit c, donc ℓ'' est aussi bijective et $\sum_{a \in B} \ell''(a) = \sum_{a \in B} \ell(a)$. Comme $\ell'' \geqslant \ell$, ceci implique $\ell'' = \ell$ d'où finalement $\ell = \ell'$.

Question 7.

Comme remarqué dans la réponse précédente, il suffit d'effectuer un parcours infixe de l'arbre et de numéroter les nœuds au fur et à mesure du parcours.

En Caml :

```
let etiquettes fils =

  let n = vect_length(fils) in
  let etiq = make_vect n 0 in

(* parcours infixe du sous-arbre de racine a, i est le num\'ero
   \`a affecter au premier noeud visit\'e, et on attend en retour
   le num\'ero \`a affecter au prochain noeud \`a visiter *)
  let rec visite a i = match fils.(a) with
    | [b;c] -> let j = visite b i in etiq.(a) <- j; visite c (j+1)
    | _     -> etiq.(a) <- i; i+1
```

```
  in
  let _ = visite 0 1 in etiq
;;
```

En Pascal :

```
procedure etiquettes(fils : vlint; var etiq : vint);
var i : integer;
  { parcours infixe du sous-arbre de racine a, i est le num\'ero
    \`a affecter au premier noeud visit\'e }
   procedure visite(a : integer);
   begin
      if fils[a] = nil then begin etiq[a] := i; i := i+1 end
      else begin
         visite(fils[a]^.val);
         etiq[a] := i;
         i := i+1;
         visite(fils[a]^.suiv^.val);
      end
   end;
begin
   i := 1;
   visite(0);
end;
```

Question 8.

$\ell(0)$ est le numéro d'ordre infixe de la racine, c'est 1 de plus que la taille de son sous-arbre gauche B_a, soit $\ell(0) = 2^h$. De même, $\ell(a) = 2^{h-1}$ et $\ell(b) = 2^h + 2^{h-1}$.

Question 9.

Soit a un nœud interne de hauteur h_a, b son fils gauche et c son fils droit. Alors $\ell(a) - \ell(b) - 1$ est la taille du sous-arbre droit de b, et $\ell(c) - \ell(a) - 1$ est la taille du sous-arbre gauche de c, soit :

$$\ell(a) - \ell(b) - 1 = \ell(c) - \ell(a) - 1 = 2^{h-h_a-1} - 1.$$

Si l'on pose $\ell(x) = 2^{h-h_x} f(x)$ on obtient $f(b) = 2f(a) - 1$, $f(c) = 2f(a) + 1$ et $f(0) = 1$ d'où par récurrence : pour tout $x \in B$, $f(x)$ est un entier impair. En particulier : *la valuation 2-adique de $\ell(x)$ est égale à $h - h_x$.*

Considérons à présent deux nœuds a, b et soit $c = \text{ppac}(a, b)$. Notons $p = \ell(a)$, $q = \ell(b)$, $r = \ell(c)$, et G, D les branches gauche et droite issues de c. Enfin, quitte à échanger a et b, supposons $p \leqslant q$. Par définition de c, on a $a \in G \cup \{c\}$ et $b \in D \cup \{c\}$ donc c est visité en ordre infixe après a et avant b, d'où $p \leqslant r \leqslant q$. De plus, si $p \leqslant s \leqslant q$ alors s est le numéro de visite d'un nœud d situé après a et avant b en ordre infixe, donc descendant de c. On en déduit que la valuation 2-adique de $\ell(d) = s$ est inférieure ou égale à celle de $\ell(c) = r$, avec égalité si et seulement si $h_d = h_c$, soit $d = c$ soit encore $s = r$.

Question 10.

D'après la question précédente, la valuation 2-adique de r est le plus grand entier $k \in \mathbb{N}$ tel que $[p, q]$ contienne un multiple de 2^k, c'est-à-dire tel que $\lceil p/2^k \rceil \leqslant \lfloor q/2^k \rfloor$. On a alors $\lceil p/2^k \rceil = \lfloor q/2^k \rfloor$ vu le caractère maximal de k, et $r = 2^k \lceil p/2^k \rceil$. On en déduit une définition récursive de μ (clairement bien fondée) :

$$\mu(p, q) = \begin{cases} p & \text{si } p = q \ ; \\ 2\mu\left(\left\lceil \frac{p}{2} \right\rceil, \left\lfloor \frac{q}{2} \right\rfloor\right) & \text{sinon.} \end{cases}$$

En Caml :

```
let rec mu p q =
  if p=q then p else 2*(mu ((p+1)/2) (q/2))
;;
```

En Pascal :

```
function mu(p,q : integer) : integer;
```

```
var s : integer;
begin
   s := 1;
   while p < q do begin p := (p+1) div 2; q := q div 2; s := 2*s end;
   mu := s*p;
end;
```

La complexité de ces deux fonctions est $O(k)$ où k est la valuation 2-adique de r, soit $k = h - h_r$. Comme a et b sont descendants de c et comme le sous-arbre de racine c contient 2^{k-1} nœuds, on a $2^{k-1} \leqslant \ell(b) - \ell(a) + 1 = q - p + 1$ d'où $k = O(\log_2(q-p))$.

Question 11.

Calcul des poids : on parcourt l'arbre en profondeur d'abord, en mettant à jour le poids d'un nœud entre chaque visite d'un fils.

En Caml :

```
let poids fils =
  let n = vect_length(fils) in
  let p = make_vect n 1 in

  (* calcule le poids du sous-arbre de racine a *)
  (* f est la liste des fils non encore examin\'es *)
  let rec calcule a f = match f with
    | [] -> ()
    | b::suite -> calcule b fils.(b); p.(a) <- p.(a) + p.(b); calcule a suite

  in calcule 0 fils.(0); p
;;
```

En Pascal (le paramètre `poids` a été renommé `w` pour éviter la confusion d'identificateurs avec le nom de la procédure) :

```
procedure poids(fils : vlint; var w : vint);
   { calcule le poids du sous-arbre de racine a }
   procedure calcule(a : integer);
   var f : lint;
   begin
      f := fils[a];
      w[a] := 1;
      while f <> nil do begin
         calcule(f^.val);
         w[a] := w[a] + w[f^.val];
         f := f^.suiv;
      end
   end;
begin
   calcule(0);
end;
```

Gauchissement : on effectue ici aussi un parcours de l'arbre (dans un ordre quelconque), au cours duquel on isole pour chaque nœud interne un fils de poids maximal pour le placer en tête de la liste des fils.

En Caml :

```
let gauchir fils w =

  (* place en t\^ete de liste un fils de poids maximum *)
  let rec reordonne f = match f with
    | [] -> []
    | a::suite -> match reordonne suite with
        | b::reste when w.(b) > w.(a) -> b::a::reste
        | l -> a::l
```

```
  in

  let n = vect_length(fils) in
  let filsG = make_vect n [] in
  for i=0 to n-1 do filsG.(i) <- reordonne fils.(i) done;
  filsG
;;
```

En Pascal :

```
procedure gauchir(fils : vlint; w : vint; var filsG : vlint);
var a,b : integer;
    f,g : lint;
begin
   for a:=0 to n-1 do begin
      f := fils[a];
      if f = nil then filsG[a] := nil
      else begin
         b := f^.val; { fils de poids maximal parmi ceux d\'ej\`a vus }
         g := nil;    { liste des autres fils d\'ej\`a vus }
         f := f^.suiv;
         while f <> nil do begin
            if w[b] > w[f^.val] then g := cons(f^.val,g)
                               else begin g := cons(b,g); b := f^.val end;
            f := f^.suiv
         end;
         filsG[a] := cons(b,g)
      end
   end
end;
```

Question 12.

Soit c le fils gauche de b. On a $c \neq a$ car a est léger, et $w(c) \geqslant w(a)$, donc $w(b) \geqslant 1+w(c)+w(a) > 2w(a)$. Soient $a = a_0, a_1, \ldots, a_k$ les ancêtres légers de a, classés par hauteur décroissante. Soit pour $i < k$, b_i le père de a_i. On a $w(a_{i+1}) \geqslant w(b_i) > 2w(a_i)$, d'où $w(a_k) > 2^k w(a)$. Or $w(a_k) \leqslant n$ et $w(a) \geqslant 1$, donc $2^k < n$ c'est-à-dire $k < \log_2(n)$. Et le nombre d'ancêtres légers de a est $1 + k < 1 + \log_2(n)$.

Question 13.

Si a est un nœud quelconque, on a :

$$\text{cime}(a) = \begin{cases} \text{pere}(a) & \text{si } a \text{ est léger ou si pere}(a) \text{ est léger ;} \\ \text{cime}(\text{pere}(a)) & \text{sinon.} \end{cases}$$

En Caml :

```
let cimes fils =
  let n = vect_length(fils) in
  let cime = make_vect n 0  in

  (* a est un noeud dont on conna\^{\i}t la cime,
     la est un bool\'een disant si a est l\'eger,
     f est la fin de la liste des fils de a,
     lf est un bool\'een disant si le premier \'el\'ement de f est l\'eger,
     calcule les cimes des \'el\'ements de f et de leurs descendants    *)
  let rec calcule a la f lf = match f with
    | [] -> ()
    | b::suite ->
        cime.(b) <- if lf or la then a else cime.(a);
        calcule b lf fils.(b) false;
        calcule a la suite true
```

```
  in calcule 0 true fils.(0) false; cime
;;
```

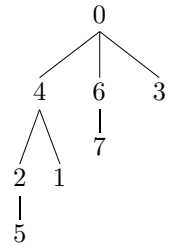
En Pascal :

```
procedure cimes(fils : vlint; var cime : vint);
  {  a est un noeud dont on conna\^{\i}t la cime,
     la est un bool\'een disant si a est l\'eger,
     calcule les cimes des descendants de a    }
   procedure calcule(a : integer; la : boolean);
   var f : lint;
      lf : boolean;
   begin
      f  := fils[a];
      lf := false;
      while f <> nil do begin
         if la or lf then cime[f^.val] := a else cime[f^.val] := cime[a];
         calcule(f^.val,lf);
         lf := true;
         f := f^.suiv
      end
   end;
begin
   cime[0] := 0; calcule(0,true);
end;
```

Question 14.

On vérifie par récurrence sur la hauteur d'un nœud a que $t_2[a]$ est la distance de a à son plus proche ancêtre léger.

$A' =$



a	0	1	2	3	4	5	6	7
$t_2[a]$	0	0	2	0	1	3	0	1
$\lambda(a)$	$\langle\rangle$	$\langle 1,1,2,0\rangle$	$\langle 1,2\rangle$	$\langle 3,0\rangle$	$\langle 1,1\rangle$	$\langle 1,3\rangle$	$\langle 2,0\rangle$	$\langle 2,0,1,1\rangle$

Question 15.

On remplit le tableau `lambda` demandé par un parcours de l'arbre en profondeur d'abord, selon le même schéma que pour le calcul des cimes.

En Caml :

```
let etiquettes fils cime =
  let n = vect_length(fils) in
  let lambda = make_vect n [] in

  (* a est un noeud, ta est la distance au plus proche anc\^etre l\'eger
     f est la fin de la liste des fils de a,
     r est le rang dans la liste des fils de a du premier \'el\'ement de f,
     calcule les \'etiquettes des \'el\'ements de f et de leurs descendants    *)
  let rec calcule a ta f r = match f with
    | [] -> ()
    | b::suite ->
        let tb = if r=1 then ta+1 else 0 in
        lambda.(b) <- lambda.(cime.(b)) @ [r;tb];
        calcule b tb fils.(b) 1;
        calcule a ta suite (r+1)
```

```
  in calcule 0 0 fils.(0) 1; lambda
;;
```

En Pascal :

```
procedure etiquettes(fils : vlint; cime : vint; var lambda : vlint);
   { a est un noeud et ta est la distance au plus proche anc\^etre l\'eger }
   procedure calcule(a : integer; ta : integer);
   var f   : lint;
       r,t : integer;
   begin
      f := fils[a];
      r := 1;
      t := ta+1;
      while f <> nil do begin
         lambda[f^.val] := concat(lambda[cime[f^.val]],cons(r,cons(t,nil)));
         calcule(f^.val,t);
         r := r+1;
         t := 0;
         f := f^.suiv
      end
   end;
begin
   lambda[0] := nil; calcule(0,0);
end;
```

Question 16.

Lemme : *soit a un nœud et $b = \mathrm{cime}(a)$. On a :*
si $t_2[a] = 0$ alors a est le $t_1[a]$-ème fils de b ;
si $t_2[a] > 0$ alors a est le $t_2[a]$-ème descendant gauche de b.

Démonstration : immédiate, par récurrence sur la hauteur de a. On en déduit un algorithme récursif d'inversion de la fonction λ (ce qui prouve accessoirement que cette fonction est injective) :

Pour calculer a tel que $\lambda(a) = \langle u_1, v_1, \ldots, u_k, v_k\rangle$:
1. si $k = 0$, alors $a = 0$.
2. sinon, soit b tel que $\lambda(b) = \langle u_1, v_1, \ldots, u_{k-1}, v_{k-1}\rangle$:
2.1. si $v_k = 0$ alors a est le u_k-ème fils de b ;
2.2 sinon a est le v_k-ème descendant gauche de b.

En Caml :

```
let trouve fils etiq =

  (* recherche r\'ecursive \`a partir du noeud x,
     e est la partie non encore utilis\'ee de l'\'etiquette *)
  let rec calcule x e = match e with
    | u::0::suite -> calcule (u_eme_fils fils.(x) u) suite
    | u::v::suite -> calcule (v_eme_gauche x v) suite
    | _           -> x
  and u_eme_fils   l u = if u = 1 then hd(l) else u_eme_fils (tl l) (u-1)
  and v_eme_gauche x v = if v = 0 then x else v_eme_gauche (hd fils.(x)) (v-1)

  in calcule 0 etiq
;;
```

En Pascal :

```
function trouve(fils : vlint; etiq : lint) : integer;
var x,u,v : integer;
    f     : lint;
begin
   x := 0;
```

```
   while etiq <> nil do begin
      u := etiq^.val;
      v := etiq^.suiv^.val;
      etiq := etiq^.suiv^.suiv;
      if v = 0 then begin
         f := fils[x];
         for u:=u downto 2 do f := f^.suiv;
         x := f^.val
      end
      else begin
         for v:=v downto 1 do x := fils[x]^.val
      end
   end;
   trouve := x
end;
```

Question 17a.

Soit a un nœud. On note $a_0 = a$, $a_1 = \text{cime}(a_0)$, $a_2 = \text{cime}(a_1)$, ... et soit k le premier entier tel que $a_k = 0$. Notons α le nombre d'éléments légers dans $(a_0, \ldots, a_k)$, β le nombre d'éléments lourds dans $(a_0, \ldots, a_k)$ et γ le nombre d'ancêtres légers de a. On a évidement $\alpha \leqslant \gamma$, et de plus, comme la cime d'un nœud lourd est toujours un nœud léger, chaque nœud lourd dans la suite $(a_0, \ldots, a_k)$ est suivi par un nœud léger, d'où $\beta \leqslant \alpha$. On en déduit $k = \alpha + \beta - 1 \leqslant 2\alpha - 1 \leqslant 2\gamma - 1$. La longueur de $\lambda(a)$ est $2k \leqslant 4\gamma - 2$.

Remarque : on vérifie par récurrence sur le nombre de nœuds légers situés entre a et b que si b est un ancêtre léger de a alors $b \in \{a_0, \ldots, a_k\}$ donc en fait $\alpha = \gamma$ ce qui implique $k + 1 \geqslant \gamma$. On en déduit : $2\gamma - 2 \leqslant \text{longueur}(\lambda(a)) \leqslant 4\gamma - 2$.

Question 17b.

Soit w le plus long préfixe de longueur paire commun à u et v. On note $u = w \circ u'$ et $v = w \circ v'$. Alors :

– si $u' = \langle 1, i\rangle \circ u''$ et $v' = \langle 1, j\rangle \circ v''$ avec $i > 0$ et $j > 0$ alors $\lambda(\text{ppac}(a, b)) = w \circ \langle 1, \min(i, j)\rangle$
– dans tous les autres cas, $\lambda(\text{ppac}(a, b)) = w$.

La justification de cet algorithme n'est pas demandée ...

* *

*