Option Informatique
TP3 : Représentations contigües d'un arbre, files de priorité, tas

Corrigé

27 novembre 2006

1 Représentations contigües d'un arbre

1.1 Utilisation d'un tableau

```
let est_terminal T =
  let r = T.racine and tab = T.sommet in
  let A = tab.(r) in
  ((A.droit = -1) && (A.gauche = -1)) ; ;

let racine T =
  if T.racine = (-1) then failwith "arbre vide"
  else ((T.sommet).(T.racine).info) ; ;

let fils_gauche T =
  racine = ((T.sommet).(T.racine)).gauche ; sommet = T.sommet ; ;

let fils_droit T =
  racine = ((T.sommet).(T.racine)).droit ; sommet = T.sommet ; ;

let cons G r D =
  let tailleg = (vect_length G.sommet)
      and tailled = (vect_length D.sommet) in
  let A = make_vect (tailleg+tailled+1)
    droit = (tailleg+D.racine) ; info = r ; gauche = G.racine in begin
    for i = 0 to (tailleg - 1) do
      A.(i+1) <- (G.sommet).(i) ;
    done ;
    for i = 0 to (tailled - 1) do
      A.(i+1+tailleg)<- (D.sommet.i) ;
    done ;
```

```
    racine = 0 ; sommet = A ;
  end ; ;
```

1.2 Une structure plus simple

Il est facile de se convaincre du bien-fondé des indices donnés pour les fils gauche et droit d'un sommet donné dans une structure de tas. On numérote les sommets d'un arbre quasi-complet selon un parcours en largeur d'abord :

– Le $d-i$ème étage comprend 2^{d-1} sommets (sauf le dernier étage, c'est l'hypothèse « quasi-complet »).
– Le k-ième sommet du d-ième étage est donc d'indice : $2^0 + 2^1 + \ldots + 2^{d-2} + k = 2^{d-1} + (k)$

Reste à montrer qu'il y a $i-1$ sommets entre un sommet d'indice i et son fils gauche (si il existe).
Et en effet, il y a :
– d'abord $2^{d-1} - k$ sommets de l'étage d (les « successeurs »),
– puis $2(k-1)$sommets de l'étage $d+1$ (les fils des prédécesseurs de l'étage d).

Soit en tout $i-1$ sommets.

```
let arbre_en_tas a h = traite_noeud 0 a
  where rec traite_noeud i = function
      Vide → ()
    | Noeud (l, n, r) →
        begin
          if i >= h.taille then h.taille <- (i+1),
          h.contenu.(i) <- n ;
          traite_noeud (2*i+1) l ;
          traite_noeud (2*i+2) r ;
        end ; ;

let tas_en_arbre h = traite_noeud 0
  where rec traite_noeud i =
  if i>=h then Vide
  else Noeud ((traite_noeud (2*i+1)), h.contenu.(i),
              (traite_noeud (2*i+2))) ; ;
```

2 files de priorité et tas

```
let echange i j t =
  let w = t.(i) in begin
    t.(i) <- t.(j) ;
    t.(j) <- w ;
  end ; ;

let insere x h = begin
  if h.taille >= vect_length h.contenu then
    failwith "débordement du tas" ;
  h.contenu.(h.taille) <- x ;
  h.taille <- h.taille +1 ;
  retablir h ;
  where retablir h =
    let dernier = ref (h.taille -1) in
    let pere = ref ((!dernier-1)/2)
    and v = h.contenu in
    while (!dernier >= 1 && v.(!pere) < v.(!dernier)) do
      echange !dernier !pere v ;
      dernier := !pere ;
      pere := (!dernier-1)/2
    done ;
end ; ;
```

La terminaison de la boucle est garantie par la stricte décroissance du contenu de la référence `dernier`, qui est divisée par 2 à chaque itération, et par l'exigence que le contenu de `dernier` est supérieur ou égal à 1 (ce qu garantit de `dernier` n'est pas la racine de l'arbre). la complexité de cette opération est donc majorée par la hauteur de l'arbre, elle est donc en $O(log_2(n))$, où n est le nombre de noeuds.

2.1 Supression dans une file de priorité

```
let supprime_racine h =
  let n = h.taille in begin
  h.contenu.(0) <- h.contenu.(n-1) ;
  h.taille <- n-1 ;
  percole h ;
  end ;
  where percole h =
      let n = h.taille and v = h.contenu in
      let rec traite_noeud i =
        if (2*i+1 <n) then
          let j = 2*i+1 in
          let fils =
            if (j+1 < n && v.(j)<v.(j+1)) then (j+1) else j
          in
            if v.(fils) < v.(i) then begin
              echange i fils v ;
              traite_noeud fils
            end
      in traite_noeud 0 ; ;
```

La percolation considère à chaque appel récursif un noeud de profondeur strictement plus grande que la profondeur du noeud de la fonction appelante. On en tire que la complexité de cette fonction est majoré par la hauteur de l'arbre, elle est donc en $O(log_2(n))$, où n est le nombre de noeuds.

2.2 Recherche du maximum dans une file de priorité

Pa rune récurrence évidente, on montre que la structure de file de priorité conserve le maximum de la file en tête de celle-ci, c'est à dire à la racine de l'arbre. Il suffit donc de faire afficher, pour un tas quelconque `h`, la valeur de `h.contenu.(0)` pour avoir la valeur de ce maximum.

2.3 Tri par tas

Le principe du tri par tas consiste à insérér successivement les éléments du tableau qu'on veut trier dans un tas initialement vide, puis on extrait itérativement la racine de ce tas (qui en est toujours le plus grand élément), jusqu'à ce que le tas soit vide.

```
let tri_en_tas v =
  let n = vect_length v in
  let h = nouveau_tas n in
    for i = 0 to (n-1) do
      insere v.(i) h
    done ;
  for i = 0 to (n-1) do
    v.(n-1) <- h.contenu.(0) ;
    supprime_racine h ;
  done ; ;
```