IFT2015 H12 — Solutionnaire de l'examen intra

Miklós Csűrös

20 février 2012

F0 Votre nom (1 point)

F1 Taux de croissance (20 points)

▶ Remplissez le tableau suivant : chaque réponse vaut 2 points. Pour chaque paire f, g, écrivez "=" si $\Theta(f) = \Theta(g)$, "$\ll$" si $f = o(g)$, "$\gg$" si $g = o(f)$, et "???" si aucun des trois cas n'applique.

$f(n)$	$g(n)$	**relation**
$f(n) = 2n^2$	$g(n) = n^2 \lg n$	$\ll$
$f(n) = \sqrt{n}$	$g(n) = \sqrt[3]{n}$	$\gg$ $n^{1/3} = o\big(n^{1/2}\big)$
$f(n) = \sum_{i=0}^{n} 2^i$	$g(n) = 2^n$	$=$ $f(n) = 2^{n+1} - 1 = (2 + o(1))2^n$
$f(n) = \sum_{i=1}^{n} 1/i$	$g(n) = \log_{2015} n$	$=$ $f(n) = H_n = \ln n + O(1)$
$f(n) = n!$	$g(n) = (2n)!$	$\ll$ $g(n) = (2n)(2n-1)\cdots(n+1) \cdot f(n)$
$f(n) = n^{2015}$	$g(n) = (2n)^{2015}$	$=$ $g(n) = n^2 015 \cdot 2^{2015} = \Theta(n^{2015})$
$f(n) = n$	$g(n) = \begin{cases} n+2 & \{n \leq 2\} \\ \frac{n \lg n}{\lg \lg n} & \{n > 2\} \end{cases}$	$\ll$ $\frac{\lg \lg n}{\lg n} = o(1)$
$f(n) = n \lg n$	$g(n) = \ln(n!)$	$=$ $g(n) = (n - \frac{1}{2}) \ln n - n + O(1)$
$f(n) = n \lg n$	$g(n) = \begin{cases} 1 & \{n = 0\} \\ 2g\big(\lfloor n/2 \rfloor\big) + \Theta(1) & \{n > 0\} \end{cases}$	$\gg$ $g(n) = \Theta(n)$
$f(n) = 2^{2^n}$	$g(n) = 4^n$	$\gg$ $f(n) = 2^{\left(2^n\right)}; g(n) = 2^{(2 \times n)}$

F2 Pair-impair (20+3 points)

On a une liste chaînée, où chaque nœud N possède les variables N.key (clé, un nombre entier) et N.next (prochain élément). On veut une procédure deleteOdd(N) qui supprime les nœuds avec clés impaires à partir de N et retourne la nouvelle tête de la liste. L'algorithme doit préserver l'ordre des nœuds qui restent.

Dans les algorithmes suivants, N.key $\cong 1$ dénote le test booléen que la clé de N est impair : N.key $\cong 1$ si et seulement si N.key $\bmod 2 = 1$.

a. Solution itérative (10 points)

```
deleteOdd(N)                                              // N = null permis
H ← N ; while H ≠ null && H.key ≅ 1 do H ← H.next         // on retournera H
E ← H                                         // E est toujours le dernier nœud pair
while E ≠ null do
    J ← E.next
    while J ≠ null && J.key ≅ 1 do J ← J.next         // chercher prochain nœud pair
    E.next ← J ; E ← J
return H
```

b. Solution récursive (10 points)

```
deleteOdd(N)
if N = null then return N
else
    if N.key ≅ 1
    then return deleteOdd(N.next)
    else N.next ← deleteOdd(N.next) ; return N
```

♡c. Récursion terminale (3 points boni)

```
deleteOdd(H, E, N)    // arguments formels correspondent aux variables locales de la version itérative
if N = null then
    if E ≠ null then E.next ← N
    return H
else
    if N.key ≅ 1 then return deleteOdd(H, E, N.next)
    else
        if E ≠ null then E.next ← N
        if H = null then return deleteOdd(N, N, N.next)
        else return deleteOdd(H, N, N.next)
```

F3 Ancêtre commun (24 points)

► Donnez un algorithme lca(u, v) qui retourne l'ACPB de deux nœuds internes u, v dans un arbre binaire.

Le plus simple est d'utiliser deux piles pour stocker les ancêtres des deux nœuds. On remplit les piles en montant à la racine, et on identifie le lca par des pop en parallèle.

```
lca(u, v)
initaliser piles P et Q
p ← u ; do P.push(p) ; p ← p.parent while p ≠ null          // ancêtres de p
q ← v ; do Q.push(q) ; q ← q.parent while q ≠ null          // ancêtres de q
p ← P.pop() ; q ← Q.pop()                                   // forcément, p = q = racine
do
    r ← p                                                   // à ce point, p = q
    if P.isEmpty() ou Q.isEmpty() then return r   // u est l'ancêtre de v ou vice verse (ou bien u = v)
    p ← P.pop() ; q ← Q.pop()
while p = q
return r
```

Temps de calcul : c'est $\Theta\bigl(d[u] + d[v]\bigr) = O(h)$ où $d[\cdot]$ dénote la profondeur de nœud et h est la hauteur de l'arbre. Notez que $h = O(\log n)$ n'est pas nécessairement vrai.

F4 Tas différentiel (35 points)

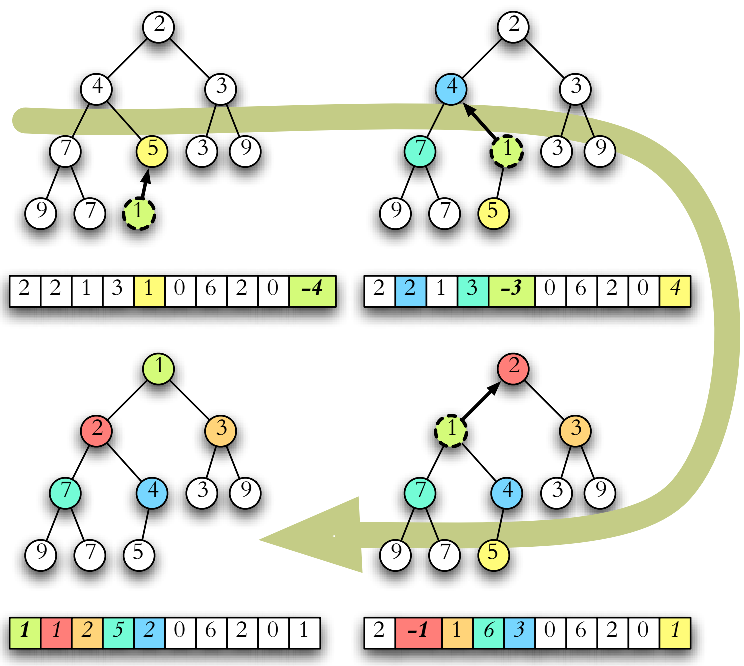
On veut une implantation de file de priorité par un tas binaire $H[1..n]$, où, à l'exception de l'élément minimal $H[1]$, on ne stocke pas la priorité des éléments directement, mais plutôt la différence de priorités entre parent et enfant. À $H[1]$, on stocke la vraie priorité de la racine.

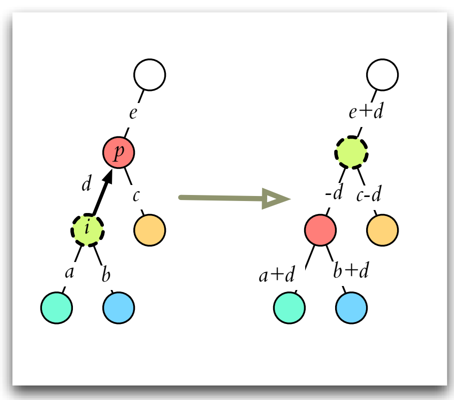
- ▶ Donnez un algorithme getPriority(i) qui calcule la vraie priorité de l'élément à l'indice i.
- ▶ Donnez un algorithme pour insérer un élément dans le tas différentiel (étant donné sa vraie priorité).

a. La vraie priorité. Il faut sommer les différences jusqu'à la racine.

getPriority(i)
P1 $x \leftarrow 0$
P2 **while** $i \neq 0$ **do** $x \leftarrow x + H[i]$; $i \leftarrow \lfloor i/2 \rfloor$
P3 **return** x

b. Insertion. Pour comprendre comment insérer, examinons comment le tas différentiel $H[]$ devrait changer quand on insère, p.e., '1'. On voit qu'en une itération de swim, jusqu'à 5 cases changent de valeur (i, ses enfants à $2i$ et $2i+1$, son parent à $\lfloor i/2 \rfloor$, et sa sœur à $i \pm 1$).



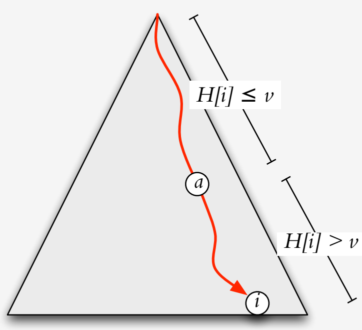


insert$(H[\,], x, n)$ — *// insertion dans le tas H à n éléments*
I1 swim$(H, n+1, x, n+1)$ — *// essai d'insertion à l'indice $n+1$*

swim(H, i, x, ℓ) — *// x est la vraie priorité*
S1 $p \leftarrow \lfloor i/2 \rfloor$; $d \leftarrow x -$ getPriority(p) — *// différence entre la priorité du parent et x*
S2 **while** $i \neq 1$ et $d < 0$ — *// boucle pour* swim
S3 $\quad H[i] \leftarrow -d$
S4 $\quad$ **if** $2p = i$ **then if** $i+1 \leq \ell$ **then** $H[i+1] \leftarrow H[i+1] - d$ — *// sœur à la droite*
S5 $\quad$ **else** $H[i-1] \leftarrow H[i-1] - d$ — *// sœur à la gauche*
S6 $\quad$ **if** $2i \leq \ell$ **then**
S7 $\quad\quad H[2i] \leftarrow H[2i] + d$ — *// enfant gauche*
S8 $\quad\quad$ **if** $2i+1 \leq \ell$ **then** $H[2i+1] \leftarrow H[2i+1] + d$ — *// enfant droit*
S9 $\quad d \leftarrow d + H[p]; i \leftarrow p; p \leftarrow \lfloor i/2 \rfloor$
S10 $H[i] \leftarrow d$

F5 ♡ Difficile à comparer. . . (12 points boni)

Dans l'implantation usuelle du tas binaire, une insertion nécessite $O(\log n)$ comparaisons entre priorités et $O(\log n)$ affectations (de cases dans le tableau du tas). Montrez comment faire l'insertion avec $O(\log \log n)$ comparaisons et $O(\log n)$ affectations.



Lors d'un appel de swim(H, i, v), on monte jusqu'à la racine pour trouver l'ancêtre a plus distante avec une priorité $H[a] > v$. On doit décaler la suite d'ancêtres vers i à partir de a, et placer v à l'indice a. On peut identifier l'ancêtre a plus rapidement, par le principe de diviser-pour-régner (comme en recherche binaire).

swim(H, i, v)
D1 $k \leftarrow 0$; $g \leftarrow i$; **while** $(g \neq 0)$ **do** $g \leftarrow \lfloor g/2 \rfloor$; $k \leftarrow k+1$
D2 $d \leftarrow i$
D3 **while** $k > 1$ — *// ancêtres $d, d/2, d/4, \ldots, d/2^{k-1}$*
D4 $\quad m \leftarrow \left\lfloor d/2^{\lfloor k/2 \rfloor} \right\rfloor$ — *// m est l'ancêtre «au milieu» : en Java, écrire* `m=d>>(k/2)`
D5 $\quad$ **if** $H[m] \leq v$ **then** $k \leftarrow \lfloor k/2 \rfloor$ — *// continuer avec ancêtres plus proches $d, d/2, d/4, \ldots, d/2^{k/2-1}$*
D6 $\quad$ **else** $d \leftarrow m$; $k \leftarrow \lceil k/2 \rceil$ — *// continuer avec ancêtres plus distantes $d/2^{k/2}, d/2^{k/2+1}, \ldots, d/2^{k-1}$*
// boucle terminée : placer v à l'indice d — il faut décaler les ancêtres entre indices d et i
D7 $p \leftarrow \lfloor i/2 \rfloor$
D8 **while** $p \neq d$ **do** $H[i] \leftarrow H[p]; i \leftarrow p; p \leftarrow \lfloor i/2 \rfloor$
D9 $H[d] \leftarrow v$