IFT2015 H12 — Examen Intra

Miklós Csűrös

20 février 2012

The English translation is on the last pages

Aucune documentation n'est permise. L'examen vaut 100 points. Vous pouvez avoir jusqu'à 15 points de boni additionnels (pour des questions dénotées par ♡).

▶ **Répondez à toutes les questions dans les cahiers d'examen.**

F0 Votre nom (1 point)

▶ Écrivez votre nom et code permanent sur tous les cahiers soumis.

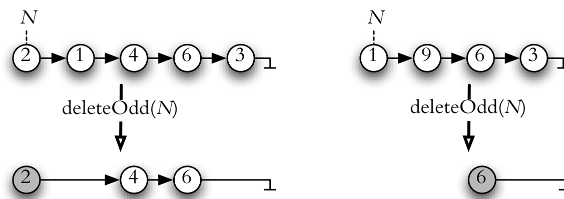
F1 Taux de croissance (20 points)

▶ 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. Il n'est pas nécessaire de justifier vos réponses. $\lg n$ dénote le logarithme binaire de n.

	$f(n)$	$g(n)$
a	$f(n) = 2n^2$	$g(n) = n^2 \lg n$
b	$f(n) = \sqrt{n}$	$g(n) = \sqrt[3]{n}$
c	$f(n) = \sum_{i=0}^{n} 2^i$	$g(n) = 2^n$
d	$f(n) = \sum_{i=1}^{n} 1/i$	$g(n) = \log_{2015} n$
e	$f(n) = n!$	$g(n) = (2n)!$
f	$f(n) = n^{2015}$	$g(n) = (2n)^{2015}$
g	$f(n) = n$	$g(n) = \begin{cases} n+2 & \{n \le 2\} \\ \frac{n \lg n}{\lg \lg n} & \{n > 2\} \end{cases}$
h	$f(n) = n \lg n$	$g(n) = \ln(n!)$
i	$f(n) = n \lg n$	$g(n) = \begin{cases} 1 & \{n = 0\} \\ 2g\bigl(\lfloor n/2 \rfloor\bigr) + \Theta(1) & \{n > 0\} \end{cases}$
j	$f(n) = 2^{2^n}$	$g(n) = 4^n$

F2 Pair-impair (20+3 points)

On a une liste chaînée, où chaque nœud N possède les variables $N.\mathsf{key}$ (clé, un nombre entier) et $N.\mathsf{next}$ (prochain élément). On veut une procédure $\mathsf{deleteOdd}(N)$ 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. Montrez tous les détails de vos algorithmes (p.e., il ne suffit pas de dire «suppression après x», il faut montrer les affectations exactes).



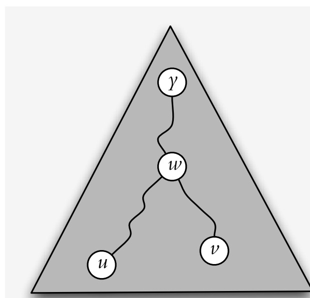
Exemples du fonctionnement de deleteOdd *: on retourne une référence au nœud ombré.*

a. Solution itérative (10 points) ► Donnez une implantation *itérative* de deleteOdd.

b. Solution récursive (10 points) ► Donnez une implantatation *récursive* de deleteOdd.

♡**c. Récursion terminale (3 points boni)** Vous aurez jusqu'à trois points boni pour une implantation avec récursion terminale en **b**.

F3 Ancêtre commun (24 points)

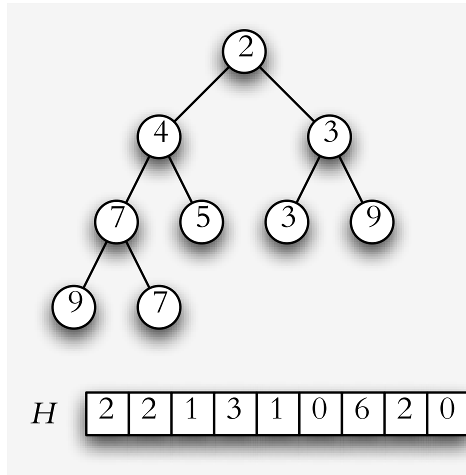


Soit u et v deux nœuds dans un arbre. Un nœud y est leur *ancêtre commun* si et seulement si u et v appartiennent au sous-arbre enraciné à y. L'*ancêtre commun le plus bas* (ACPB) est l'ancêtre w tel que son sous-arbre ne contient pas d'autres ancêtres communs.

► Donnez un algorithme $\mathsf{lca}(u, v)$ qui retourne l'ACPB de deux nœuds internes u, v dans un arbre binaire. Analysez le temps de calcul de l'algorithme.

Indice. Rassemblez les ancêtres de u et v dans une structure de données convéniente, et identifiez les ancêtres en commun.

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. (L'interface de la file de priorité reste le même. Une telle implantation permet l'ajustement de toutes les priorités par la même valeur en $O(1)$ — il suffit de changer la priorité à la racine.)

a. La vraie priorité (10 points). ► Donnez un algorithme getPriority(i) qui calcule la vraie priorité de l'élément à l'indice i. ($1 \le i \le n$)

b. Insertion (25 points). ► Donnez un algorithme pour insérer un élément dans le tas différentiel (étant donné sa vraie priorité). L'insertion doit toujours prendre $O(\log n)$.

Indice. Ici, on doit adapter la procédure de swim. D'abord, calculez la différence entre parent et enfant au point d'insertion (dernière case du tas) — cela peut-être négative indiquant la nécessité de monter vers la racine. Examinez comment les différences entre priorités changent lors d'une itération de swim.

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. (Une telle solution est utile quand la comparaison est plus couteuse que l'affectation — p.e., avec des chaînes de caractères.)

BONNE CHANCE !

English translation

No documentation is allowed. The examen is worth 100 points, and you can collect up to 15 additional bonus points for exercises marked by ♡. You may write your answers in English or in French.

Answer each question in the exam booklet.

E0 Your name (1 point)

► Write your name and *code permanent* on each booklet that you submit.

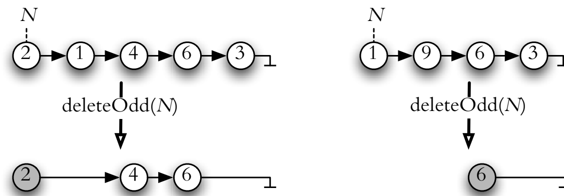
E1 Growth rates (20 points)

► Fill out the following table : every answer is worth 2 points. For each pair f, g, write "=" if $\Theta(f) = \Theta(g)$, "$\ll$" if $f = o(g)$, "$\gg$" if $g = o(f)$, and "???" if neither of the three applies. You do not need to justify the answers. $\lg n$ denotes the binary logarithm of n.

	$f(n)$	$g(n)$
a	$f(n) = 2n^2$	$g(n) = n^2 \lg n$
b	$f(n) = \sqrt{n}$	$g(n) = \sqrt[3]{n}$
c	$f(n) = \sum_{i=0}^{n} 2^i$	$g(n) = 2^n$
d	$f(n) = \sum_{i=1}^{n} 1/i$	$g(n) = \log_{2015} n$
e	$f(n) = n!$	$g(n) = (2n)!$
f	$f(n) = n^{2015}$	$g(n) = (2n)^{2015}$
g	$f(n) = n$	$g(n) = \begin{cases} n+2 & \{n \leq 2\} \\ \frac{n \lg n}{\lg \lg n} & \{n > 2\} \end{cases}$
h	$f(n) = n \lg n$	$g(n) = \ln(n!)$
i	$f(n) = n \lg n$	$g(n) = \begin{cases} 1 & \{n = 0\} \\ 2g\big(\lfloor n/2 \rfloor\big) + \Theta(1) & \{n > 0\} \end{cases}$
j	$f(n) = 2^{2^n}$	$g(n) = 4^n$

E2 Even-odd (20+3 points)

We have a linked list where every node N is equipped with the variables N.key (an integer key) and N.next (next node). We would like to have a procedure called deleteOdd(N) that deletes the nodes with odd keys on the list starting with N, and returns the new head of the list. The algorithm must keep the original order of the nodes with even keys. Show all the details (e.g., it is not enough to say "deletion after x :" you need to show the exact instructions).



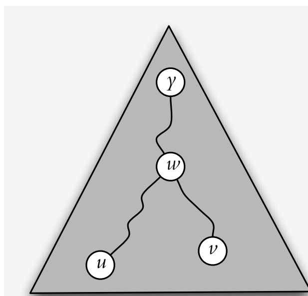
Examples of deleteOdd *: the shaded nodes indicate the return value.*

a. Iteration (10 points) ► Give an *iterative* implementation of deleteOdd.

b. Recursion (10 points) ► Give a *recursive* implementation of deleteOdd.

♡**c. Terminal recursion (3 bonus points)** You can have up to 3 bonus points for terminal recursion in **b**.

E3 Common ancestor (24 points)

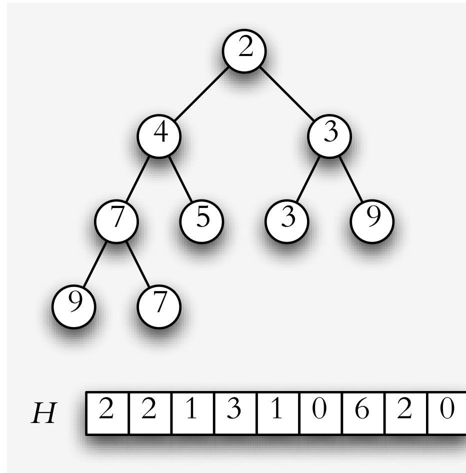


Let u and v be two nodes in a tree. A node y is their *common ancestor* if and only if u et v belong to the subtree rooted at y. The *lowest common ancestor* (LCA) is the ancestor w such that its subtree contains no other ancestor.

► Give an algorithm lca(u, v) that returns the LCA of two internal nodes u, v in a binary tree. Analyze your algorithm's running time.

Hint. Collect the ancestors of u and v in a convenient data structure, and identify the common ones.

E4 Differential heap (35 points)



We would like to implement a priority queue by using a binary heap where, with the exception of the minimal element $H[1]$, the priorities are not stored directly. Instead, we store the difference between the priorities of parent and child. At $H[1]$, we store the true priority. (The priority queue interface is not changed. This kind of implementation is useful for ajusting all priorities by the same amount in $O(1)$ — it is enough to change the priority at the root.)

a. The true priorities (10 points). ► Give an algorithm $\mathsf{getPriority}(i)$ that computes the true priority of an element stored at index i. ($1 \leq i \leq n$)

b. Insertion (25 points). ► Give an algorithm for inserting an element in a differential heap, given its true priority.

Hint. Here, you need to adapt the classic swim procedure. First, calculate the difference between parent and child at the point of insertion (last entry in the heap's array) — it may be negative, indicating that one should move up towards the root. Examine how differences between priorities should change in one iteration of swim.

E5 ♡ Difficult to compare. . . (12 bonus points)

In the usual implementaion of the binary heap, one insertion uses $O(\log n)$ comparisons between keys, and $O(\log n)$ assignments (to a table cell). Show how to do the insertion with $O(\log \log n)$ comparisons. (Such a solution may be useful when comparison is much more expensive than assignment — e.g., as with strings.)

GOOD LUCK !