Nom :

Prénom :

Examen d'algorithmique

EPITA ING1 2013 S1; A. DURET-LUTZ

Durée : 1 heure 30

Janvier 2011

Consignes

- Cet examen se déroule **sans document** et **sans calculatrice**.
- Répondez sur le sujet dans les cadres prévus à cet effet.
- Il y a 5 pages d'énoncé, et une page d'annexe.
  Rappelez votre nom en haut de chaque feuille au cas où elles se mélangeraient.
- Ne donnez pas trop de détails. Lorsqu'on vous demande des algorithmes, on se moque des points-virgules de fin de commande etc. Écrivez simplement et lisiblement. Des spécifications claires et implémentables sont préférées à du code C ou C++ verbeux.
- Le barème est indicatif et correspond à une note sur 25.

1 Dénombrement (2 pts)

```
for (int i = 0; i <= N; i += 2)
  for (int j = i; j > 0; --j)
    puts("x");
```

Combien de fois le programme ci-dessus affiche-t-il `"x"` ? Donnez votre réponse en fonction de N en distinguant le cas où N est pair de celui où N est impair.

Réponse :

2 Ordres de grandeur (2 pts)

Lesquelles de ces affirmations sont vraies ?

□ $(\sqrt{n})^n \in \mathrm{O}(n^{\sqrt{n}})$ □ $(\sqrt{n})^n \in \Omega(n^{\sqrt{n}})$ □ $n^{\sqrt{n}} \in \Theta((\sqrt{n})^n)$ □ $\log_2(n!) \in \mathrm{O}(n \log n)$

□ $n^{\sqrt{n}} \in \mathrm{O}((\sqrt{n})^n)$ □ $n^{\sqrt{n}} \in \Omega((\sqrt{n})^n)$ □ $\log_2(n!) \in \Theta(n \ln n)$ □ $\log_2(n!) \in \Omega(n^n)$

3 File de Priorité à double entrée (6 pts)

Une file de priorité à double entrée est un type de données abstrait représentant un ensemble de valeurs supportant les opérations suivantes :

- $x \leftarrow$ FINDMAX(S) retourne la valeur maximale de S
- DELETEMAX(S) retire la valeur maximale de S (cela inclut la recherche de cette valeur)
- $x \leftarrow$ FINDMIN(S) retourne la valeur minimale S
- DELETEMIN(S) retire la valeur minimale de S (cela inclut la recherche de cette valeur)
- INSERT(S,x) insère la valeur x dans S

Une telle interface peut être implémentée sur plusieurs structures de données. Indiquez la complexité de ces cinq opérations sur chacune des structures de donnéees proposées dans le tableau suivant. Utilisez les notations Θ et O de façon précise, et en fonction du nombre n d'éléments présents dans la file de priorité.

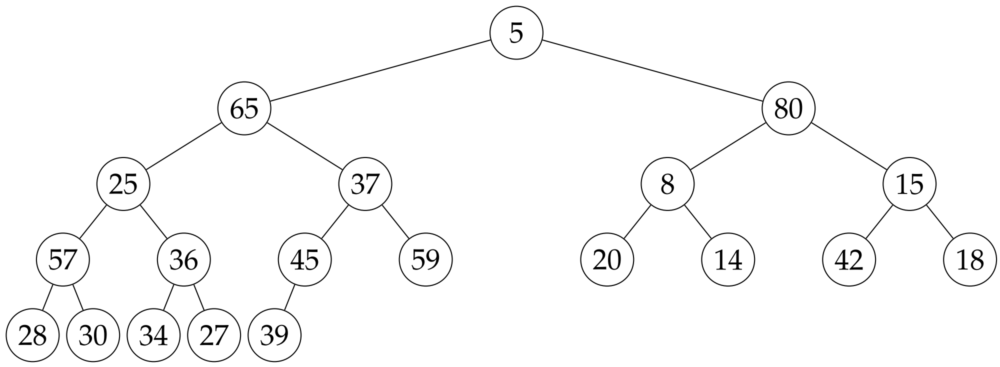
	FINDMAX	DELETEMAX	FINDMIN	DELETEMIN	INSERT
Un tableau non trié					
Un tableau circulaire trié (valeurs croissantes)					
Une liste simplement chaînée triée (valeurs croissantes) sans pointeur sur la queue					
Une liste doublement chaînée, circulaire et triée					
Un tas "max" (c'est-à-dire dont chaque nœud est plus grand que ses fils)					
Un tas "min" (c'est-à-dire dont chaque nœud est plus petit que ses fils)					
Un tas "min" et un tas "max" maintenus en parallèle (insertion et suppression des valeurs se font dans les deux tas)					
Une arbre rouge et noir					

4 Tas Min-Max (10 pts)

Un *Tas Min-Max* est une structure de donnée hybride entre le *Tas Min* et le *Tas Max*. Plus précisément, il s'agit d'un arbre binaire parfait dans lequel :

– les nœuds à un niveau **pair** sont plus **petits** que leurs descendants (directs ou indirects), et
– les nœuds à un niveau **impair** sont plus **grands** que leurs descendants (directs ou indirects).

Voici un exemple de tas min-max :



Constatez que la racine possède forcément la valeur minimale du tas, et que la valeur maximale est forcément l'un des deux fils de la racine (ou la racine elle-même si elle n'a pas de fils).

L'arbre étant parfait, il sera représenté et manipulé sous la forme d'un tableau :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
5	65	80	25	37	8	15	57	36	45	59	20	14	42	18	28	30	34	27	39

Les fils du nœud d'indice i se trouvent aux positions $\text{LEFTCHILD}(i) = 2i$ et $\text{RIGHTCHILD}(i) = 2i + 1$ lorsqu'elles existent. Le père est à la position $\text{PARENT}(i) = \lfloor i/2 \rfloor$ si elle existe.

1. **(2 pts)** Écrivez une fonction ISONMINLEVEL(A,i) qui décide en temps constant si l'indice i correspond à un nœud se trouvant sur un niveau pair (ou niveau "min") du tas min-max représenté par le tableau A. Les indices dans A commencent à 1 et vous pouvez supposer que le calcul d'un logarithme se fait un temps constant.

Réponse :

Les algorithmes sur les *tas min-max* sont similaires aux algorithmes sur les *tas max* vus en cours.

La procédure HEAPIFY(A,i), prend un nœud d'indice i dont les deux sous-arbres vérifient la propriété de tas, et fait redescendre la valeur $A[i]$ dans l'un des sous-arbres afin que la propriété du tas soit vérifiée à partir de l'indice i. Dans le cas des *tas min-max*, cette procédure est obligée de distinguer entre les niveaux *min* et les niveaux *max* comme suit :

HEAPIFY(A,i)
  if ISONMINLEVEL(A,i)
    then HEAPIFYMIN(A,i)
    else HEAPIFYMAX(A,i)

HEAPIFYMIN(A,i)
if $A[i]$ has children **then**
  $m \leftarrow$ index of the smallest of the children and grandchildren (if any) of $A[i]$
  if $A[m] < A[i]$ **then**
    $A[m] \leftrightarrow A[i]$
    if $A[m]$ is a grandchild of $A[i]$ **then**
      if $A[m] > A[\text{PARENT}(m)]$ **then** $A[m] \leftrightarrow A[\text{PARENT}(m)]$
      HEAPIFYMIN(A,m)

La procédure HEAPIFYMAX est identique à HEAPIFYMIN en changeant le sens des comparaisons et en définissant m comme l'indice du plus grand des fils et petits-fils.

2. **(2 pts)** Donnez la complexité en pire cas de HEAPIFY en fonction du nombre n de valeurs contenues dans A. Justifiez votre réponse.

Réponse :

Supprimer la plus petite valeur (la racine) ou la plus grande valeur (l'un de ses fils s'ils existent) d'un *tas min-max* se fait comme sur les tas classiques en remplaçant cette valeur par la dernière du tableau (dont on réduit la taille de 1) et en appelant HEAPIFY sur le nœud dont la valeur vient d'être modifiée.

3. **(3 pts)** Écrivez la procédure DELETEMAX(A) qui supprime la valeur maximale du *tas min-max* représenté par le tableau A. Vous pouvez supposer que le tableau possède un champ $A.size$ donnant sa taille, et que cette procédure n'est jamais appelée sur un tableau vide.

Réponse :

4. **(1 pt)** Quelle est la complexité de la procédure précédente en fonction de la taille n du tableau A ? (Vous pouvez exprimer cette complexité en fonction de la complexité $H(n)$ de HEAPIFY si vous n'avez pas répondu à la question 2.)

Réponse :

5. **(2 pts)** Donnez le contenu du tableau A de l'exemple après **deux** appels à DELETEMAX(A).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Nous ne traitons pas l'insertion dans cet examen, mais la procédure est semblable à celle d'un tas classique : on ajoute en feuille puis on fait remonter la valeur pour respecter les contraintes.

5 Diviser pour régner (5 pts)

Soit un gros problème de taille n, assez compliqué pour que vous n'ayez pas envie d'en connaître les détails. Pour résoudre ce problème, on vous propose trois algorithmes :

L'algorithme A résout le problème en le divisant en 5 sous-problèmes de taille $n/2$, en résolvant ces cinq sous problèmes récursivement, puis en combinant les solutions en temps linéaire.

L'algorithme B résout le problème en résolvant récursivement deux problème de taille $n-1$ puis en combinant leurs solutions en temps constant.

L'algorithme C résout le problème en le divisant en 9 sous-problèmes de taille $n/3$, en résolvant ces neuf sous-problèmes récursivement, puis en combinant les solutions en $\Theta(n^2)$.

Dans les trois cas, le découpage du problème en sous-problèmes se fait en temps constant.

Calculez la complexité de chacun de ces algorithmes.

Réponse :

Notations asymptotiques

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$
$$\Omega(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$
$$\Theta(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^{+\star}, \exists c_2 \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff g(n) \in \mathrm{O}(f(n)) \text{ et } f(n) \notin \mathrm{O}(g(n)) \qquad f(n) \in \mathrm{O}(g(n)) \iff g(n) \in \Omega(f(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f(n) \in \mathrm{O}(g(n)) \text{ et } g(n) \notin \mathrm{O}(f(n)) \qquad f(n) \in \Theta(g(n)) \iff \begin{cases} f(n) \in \Omega(g(n)) \\ g(n) \in \Omega(f(n)) \end{cases}$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^{+\star} \iff f(n) \in \Theta(g(n))$$

Ordres de grandeurs

constante	$\Theta(1)$	
logarithmique	$\Theta(\log n)$	
polylogarith.	$\Theta((\log n)^c)$	$c > 1$
	$\Theta(\sqrt{n})$	
linéaire	$\Theta(n)$	
	$\Theta(n \log n)$	
quadratique	$\Theta(n^2)$	
	$\Theta(n^c)$	$c > 2$
exponentielle	$\Theta(c^n)$	$c > 1$
factorielle	$\Theta(n!)$	
	$\Theta(n^n)$	

Identités utiles

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$
$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} \qquad \text{si } x \neq 1$$
$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad \text{si } |x| < 1$$
$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad \text{si } |x| < 1$$
$$\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$$
$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Définitions diverses

La complexité d'un problème est celle de l'algorithme le plus efficace pour le résoudre.

Un tri stable préserve l'ordre relatif de deux éléments égaux (au sens de la relation de comparaison utilisée pour le tri).

Un tri en place utilise une mémoire temporaire de taille constante (indépendante de n).

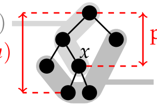
Théorème général

Soit à résoudre $T(n) = aT(n/b) + f(n)$ avec $a \geq 1$, $b > 1$

- Si $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ pour un $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.
- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour un $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour un $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$.

(Note : il est possible de n'être dans aucun de ces trois cas.)

Arbres



(nœuds $n = ni + f$)

Pour tout arbre binaire :

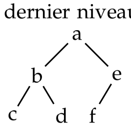
$n \leq 2^{h+1} - 1$ $\qquad h \geq \lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$ si $n > 0$

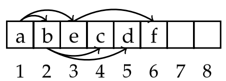
$f \leq 2^h$ $\qquad h \geq \lceil \log_2 f \rceil$ si $f > 0$

$f = ni + 1$ (si l'arbre est *complet* = les nœud internes ont tous 2 fils)

Dans un arbre binaire équilibré une feuille est soit à la profondeur $\lfloor \log_2(n+1) - 1 \rfloor$ soit à la profondeur $\lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$. Pour ces arbres $h = \lfloor \log_2 n \rfloor$.

Un arbre parfait (= complet, équilibré, avec toutes les feuilles du dernier niveau à gauche) **étiqueté** peut être représenté par un tableau.





Les indices sont reliés par :

$\text{Père}(y) = \lfloor y/2 \rfloor$

$\text{FilsG}(y) = y \times 2$

$\text{FilsD}(y) = y \times 2 + 1$

Rappels de probabilités

Espérance d'une variable aléatoire X : C'est sa valeur attendue, ou moyenne. $\mathrm{E}[X] = \sum_x \Pr\{X = x\}$

Variance : $\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - \mathrm{E}^2[X]$

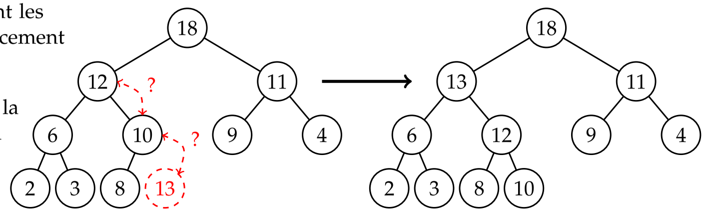
Loi binomiale : On lance n ballons dans r paniers. Les chutes dans les paniers sont équiprobables ($p = 1/r$). On note X_i le nombre de ballons dans le panier i. On a $\Pr\{X_i = k\} = \mathrm{C}_n^k p^k (1-p)^{n-k}$. On peut montrer $E[X_i] = np$ et $\mathrm{Var}[X_i] = np(1-p)$.

Tas

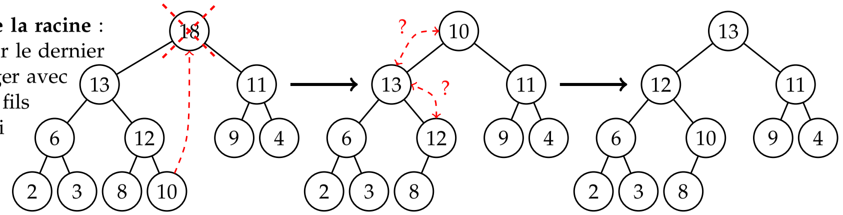
Un tas est un arbre parfait partiellement ordonné : l'étiquette d'un nœud est supérieure à celles de ses fils.

Dans les opérations qui suivent les arbres parfaits sont plus efficacement représentés par des tableaux.

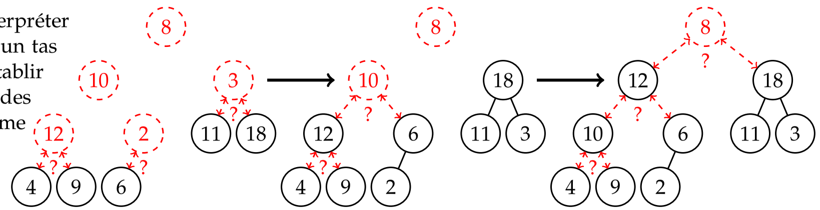
Insertion : ajouter l'élément à la fin du tas, l'échanger avec son père tant qu'il lui est inférieur (en remontant vers la racine). $T_{\text{insert}} = \mathrm{O}(\log n)$



Suppression de la racine : la remplacer par le dernier nœud ; l'échanger avec son plus grand fils tant que celui-ci est plus grand. $T_{\text{rem}} = \mathrm{O}(\log n)$



Construction : interpréter le tableau comme un tas (incorrect) puis rétablir l'ordre en partant des feuilles (vues comme des tas corrects). $T_{\text{build}} = \Theta(n)$

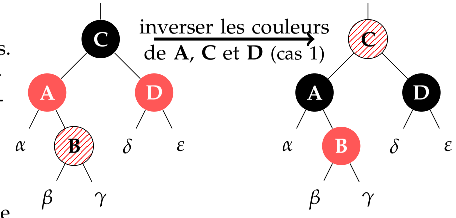


Arbres Rouge et Noir

Les ARN sont des arbres binaires de recherche dans lesquels : (1) un nœud est rouge ou noir, (2) racine et feuilles (NIL) sont noires, (3) les fils d'un nœud rouge sont noirs, et (4) tous les chemins reliant un nœud à une feuille (de ses descendants) contiennent le même nombre de nœuds noirs (= la *hauteur noire*). Ces propriétés interdisent un trop fort déséquilibre de l'arbre, sa hauteur reste en $\Theta(\log n)$.

Insertion d'une valeur : insérer le nœud avec la couleur rouge à la position qu'il aurait dans un arbre binaire de recherche classique, puis, si le père est rouge, considérer les trois cas suivants dans l'ordre.

Cas 1 : Le père et l'oncle du nœud considéré sont tous les deux rouges. **Répéter** cette transformation à partir du grand-père si l'arrière grand-père est aussi rouge.



(Dans tous ces cas, les lettres grecques représentent des sous-arbres avec la même hauteur noire.)

Cas 2 : Si le père est rouge, l'oncle noir, et que le nœud courant n'est pas dans l'axe père–grand-père, une rotation permet d'aligner fils, père, et grand-père.

Cas 3 : Si le père est rouge, l'oncle noir, et que le nœud courant est dans l'axe père–grand-père, un rotation et une inversion de couleurs rétablissent les propriétés des ARN.