ÉCOLE POLYTECHNIQUE – ÉCOLES NORMALES SUPÉRIEURES

CONCOURS D'ADMISSION 2014 **FILIÈRE MP SPÉCIALITÉ INFO**

COMPOSITION D'INFORMATIQUE – A – (XULCR)

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Arbres croissants

On étudie dans ce problème la structure d'*arbre croissant*, une structure de données pour réaliser des files de priorité.

La partie I introduit la notion d'arbre croissant et la partie II les opérations élémentaires sur les arbres croissants. L'objet de la partie III est l'analyse des performances de ces opérations. Enfin la partie IV applique la structure d'arbre croissant, notamment au problème du tri.

Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes. Les tableaux sont indexés à partir de 0, indépendamment du langage de programmation choisi. On note $\log(n)$ le logarithme à base 2 de n.

Arbres binaires. Dans ce problème, on considère des arbres binaires. Un arbre est soit l'arbre vide, noté $\mathtt{E}$, soit un nœud constitué d'un sous-arbre gauche g, d'un entier x et d'un sous-arbre droit d, noté $\mathtt{N}(g,x,d)$. La *taille* d'un arbre t, notée $|t|$, est définie récursivement de la manière suivante :

$$\begin{aligned} |\mathtt{E}| &= 1 \\ |\mathtt{N}(g,x,d)| &= 1+|g|+|d|. \end{aligned}$$

La *hauteur* d'un arbre t, notée $h(t)$, est définie récursivement de la manière suivante :

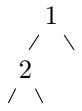
$$\begin{aligned} h(\mathtt{E}) &= 0 \\ h(\mathtt{N}(g,x,d)) &= 1+\max(h(g),\, h(d)). \end{aligned}$$

Le nombre d'occurrences d'un entier y dans un arbre t, noté $occ(y,t)$, est défini récursivement de la manière suivante :

$$\begin{aligned} occ(y,\mathtt{E}) &= 0 \\ occ(y,\mathtt{N}(g,x,d)) &= 1+occ(y,g)+occ(y,d) \quad \text{si } y=x \\ occ(y,\mathtt{N}(g,x,d)) &= occ(y,g)+occ(y,d) \quad \text{sinon.} \end{aligned}$$

L'ensemble des éléments d'un arbre t est l'ensemble des entiers y pour lesquels $occ(y,t) > 0$.

Par la suite, on s'autorisera à dessiner les arbres de la manière usuelle. Ainsi l'arbre $\mathtt{N}(\mathtt{N}(\mathtt{E},2,\mathtt{E}),1,\mathtt{E})$ pourra être dessiné sous la forme



On se donne le type `arbre` suivant pour représenter les arbres binaires.

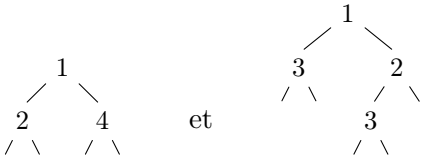
```
          (* Caml *)
type arbre =
E | N of arbre * int * arbre;;
```

```
                    { Pascal }
type arbre = ^noeud;
noeud = record gauche: arbre;
element: integer; droit: arbre; end;
```

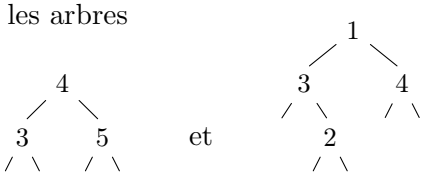
En Pascal, l'arbre vide est la constante `const E: arbre = nil` et on suppose donnée une fonction `function N(g: arbre; x: integer; d: arbre) : arbre;` pour construire le nœud $\mathtt{N}(\mathtt{g},\mathtt{x},\mathtt{d})$.

Partie I. Structure d'arbre croissant

On dit qu'un arbre t est un *arbre croissant* si, soit $t = \mathtt{E}$, soit $t = \mathtt{N}(g,x,d)$ où g et d sont eux-mêmes deux arbres croissants et x est inférieur ou égal à tous les éléments de g et d. Ainsi les arbres



sont des arbres croissants mais les arbres



n'en sont pas.

Question 1 Écrire une fonction `minimum` qui prend en argument un arbre croissant t, en supposant $t \neq \mathtt{E}$, et renvoie son plus petit élément.

```
(* Caml *)  minimum: arbre -> int
{ Pascal }  function minimum(t: arbre) : integer;
```

Question 2 Écrire une fonction `est_un_arbre_croissant` qui prend en argument un arbre t et détermine s'il a la structure d'arbre croissant. On garantira une complexité $O(|t|)$.

```
(* Caml *)  est_un_arbre_croissant: arbre -> bool
{ Pascal }  function est_un_arbre_croissant(t: arbre) : boolean;
```

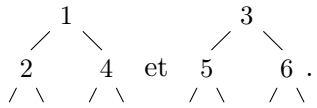
Question 3 Montrer qu'il y a exactement $n!$ arbres croissants possédant n nœuds étiquetés par les entiers $1, \ldots, n$ (chaque nœud étant étiqueté par un entier distinct).

Partie II. Opérations sur les arbres croissants

L'opération de *fusion* de deux arbres croissants t_1 et t_2, notée $\texttt{fusion}(t_1, t_2)$, est définie par récurrence de la manière suivante :

$$\begin{array}{rcll}
\texttt{fusion}(t_1, \texttt{E}) & = & t_1 & \\
\texttt{fusion}(\texttt{E}, t_2) & = & t_2 & \\
\texttt{fusion}(\texttt{N}(g_1, x_1, d_1), \texttt{N}(g_2, x_2, d_2)) & = & \texttt{N}(\texttt{fusion}(d_1, \texttt{N}(g_2, x_2, d_2)), x_1, g_1) & \text{si } x_1 \leq x_2 \\
\texttt{fusion}(\texttt{N}(g_1, x_1, d_1), \texttt{N}(g_2, x_2, d_2)) & = & \texttt{N}(\texttt{fusion}(d_2, \texttt{N}(g_1, x_1, d_1)), x_2, g_2) & \text{sinon.}
\end{array}$$

Note importante : dans la troisième (resp. la quatrième) ligne de cette définition, on a sciemment échangé les sous-arbres g_1 et d_1 (resp. g_2 et d_2). Dans les parties III et IV de ce problème apparaîtront les avantages de la fusion telle que réalisée ci-dessus (d'autres façons de réaliser la fusion n'auraient pas nécessairement de telles propriétés).

Question 4 Donner le résultat de la fusion des arbres croissants  et .

Question 5 Soit t le résultat de la fusion de deux arbres croissants t_1 et t_2. Montrer que t possède la structure d'arbre croissant et que, pour tout entier x, $occ(x, t) = occ(x, t_1) + occ(x, t_2)$.

Question 6 Déduire de l'opération `fusion` une fonction `ajoute` qui prend en arguments un entier x et un arbre croissant t et renvoie un arbre croissant t' tel que $occ(x, t') = 1 + occ(x, t)$ et $occ(y, t') = occ(y, t)$ pour tout $y \neq x$.

```
(* Caml *)  ajoute: int -> arbre -> arbre
{ Pascal }  function ajoute(x: integer; t: arbre) : arbre;
```

Question 7 Déduire de l'opération `fusion` une fonction `supprime_minimum` qui prend en argument un arbre croissant t, en supposant $t \neq \texttt{E}$, et renvoie un arbre croissant t' tel que, si m désigne le plus petit élément de t, on a $occ(m, t') = occ(m, t) - 1$ et $occ(y, t') = occ(y, t)$ pour tout $y \neq m$.

```
(* Caml *)  supprime_minimum: arbre -> arbre
{ Pascal }  function supprime_minimum(t: arbre) : arbre;
```

Question 8 Soient $x_0, \ldots, x_{n-1}$ des entiers et $t_0, \ldots, t_n$ les $n+1$ arbres croissants définis par $t_0 = \texttt{E}$ et $t_{i+1} = \texttt{fusion}(t_i, \texttt{N}(\texttt{E}, x_i, \texttt{E}))$ pour $0 \leq i < n$. Écrire une fonction `ajouts_successifs` qui

prend en argument un tableau contenant les entiers $x_0, \ldots, x_{n-1}$ et qui renvoie l'arbre croissant t_n.

```
(* Caml *)  ajouts_successifs: int vect -> arbre
{ Pascal }  function ajouts_successifs(x: array[0..n-1] of integer) : arbre;
```

Question 9 Avec les notations de la question précédente, donner, pour tout n, des valeurs $x_0, \ldots, x_{n-1}$ qui conduisent à un arbre croissant t_n de hauteur au moins égale à $n/2$.

Question 10 Toujours avec les notations de la question 8, donner la hauteur de l'arbre t_n obtenu à partir de la séquence d'entiers $1, 2, \ldots, n$, c'est-à-dire $x_i = i + 1$. On justifiera soigneusement la réponse.

Partie III. Analyse

On dit qu'un nœud $\mathtt{N}(g, x, d)$ est *lourd* si $|g| < |d|$ et qu'il est *léger* sinon. On définit le *potentiel* d'un arbre t, noté $\Phi(t)$, comme le nombre total de nœuds lourds qu'il contient.

Question 11 Écrire une fonction `potentiel` qui prend en argument un arbre t et renvoie $\Phi(t)$, tout en garantissant une complexité $O(|t|)$.

```
(* Caml *)  potentiel: arbre -> int
{ Pascal }  function potentiel(t: arbre) : integer;
```

On définit le *coût* de la fusion des arbres croissants t_1 et t_2, noté $C(t_1, t_2)$, comme le nombre d'appels récursifs à la fonction `fusion` effectués pendant le calcul de $\mathtt{fusion}(t_1, t_2)$. En particulier, on a $C(t, \mathtt{E}) = C(\mathtt{E}, t) = 0$.

Question 12 Soient t_1 et t_2 deux arbres croissants et t le résultat de $\mathtt{fusion}(t_1, t_2)$. Montrer que

$$C(t_1, t_2) \leq \Phi(t_1) + \Phi(t_2) - \Phi(t) + 2(\log |t_1| + \log |t_2|). \tag{1}$$

Question 13 Soient $x_0, \ldots, x_{n-1}$ des entiers et $t_0, \ldots, t_n$ les $n + 1$ arbres croissants définis par $t_0 = \mathtt{E}$ et $t_{i+1} = \mathtt{fusion}(t_i, \mathtt{N}(\mathtt{E}, x_i, \mathtt{E}))$ pour $0 \leq i < n$. Montrer que le coût total de cette construction est en $O(n \log(n))$.

Question 14 Montrer que, dans la construction de la question précédente, une des opérations `fusion` peut avoir un coût au moins égal à $n/2$ (on exhibera des valeurs $x_0, \ldots, x_{n-1}$ le mettant en évidence). Justifier alors la notion de complexité *amortie* logarithmique pour la fusion de deux arbres croissants.

Question 15 Soit t_0 un arbre croissant contenant n nœuds, c'est-à-dire de taille $2n + 1$. On construit alors les n arbres croissants $t_1, \ldots, t_n$ par $t_{i+1} = \mathtt{fusion}(g_i, d_i)$ où $t_i = \mathtt{N}(g_i, x_i, d_i)$, pour $0 \leq i < n$. En particulier, on a $t_n = \mathtt{E}$. Montrer que le coût total de cette construction est en $O(n \log(n))$.

Partie IV. Applications

Question 16 En utilisant la structure d'arbre croissant, écrire une fonction `tri` qui trie un tableau `a` de n entiers, dans l'ordre croissant, en temps $O(n\log(n))$. Ainsi, si le tableau `a` contient initialement $[3;2;7;2;4]$, il devra contenir $[2;2;3;4;7]$ après l'appel à la fonction `tri`. La fonction `tri` pourra allouer autant de structures intermédiaires que nécessaire, en particulier des arbres croissants. On justifiera soigneusement la complexité.

```
(* Caml *)  tri: int vect -> unit
{ Pascal }  procedure tri(a: array[0..n-1] of integer);
```

Soient $x_0, \ldots, x_{n-1}$ des entiers, avec $n = 2^k$ et $k \geq 0$. On définit une famille d'arbres croissants t_i^j, avec $0 \leq i \leq k$ et $0 \leq j < 2^{k-i}$, de la façon suivante :

$$\begin{cases} t_0^j &= \texttt{N}(\texttt{E}, x_j, \texttt{E}) & \text{pour } 0 \leq j < 2^k, \\ t_{i+1}^j &= \texttt{fusion}(t_i^{2j}, t_i^{2j+1}) & \text{pour } 0 \leq i < k \text{ et } 0 \leq j < 2^{k-i-1}. \end{cases}$$

Question 17 Montrer que le coût total de la construction de tous les arbres croissants t_i^j est en $O(2^k)$.

Question 18 En déduire une fonction `construire` qui prend en argument un tableau `a` de taille n, avec $n = 2^k$ et $k \geq 0$, et renvoie un arbre croissant contenant les éléments de `a` en temps $O(n)$.

```
(* Caml *)  construire: int vect -> arbre
{ Pascal }  function construire(a: array[0..n-1] of integer) : arbre;
```

Question 19 Comment relâcher la contrainte $n = 2^k$ pour traiter le cas d'un nombre quelconque d'éléments, toujours en temps $O(n)$? Donner le programme correspondant.

Note : Ces tas auto-équilibrés sont appelés « skew heaps » en anglais. Outre leur caractère persistant, ils offrent l'une des solutions les plus simples pour obtenir un tri de complexité optimale.

* *
*