Devoir surveillé #4

Vendredi 2026-02-27 ; durée : quatre heures

Ce sujet est adapté de l'épreuve de *Composition d'Informatique A* du concours d'admission X-ENS 2014. Les seules différences avec le sujet original sont des modifications mineures du texte, principalement dues au changement des langage de programmation enseignés en CPGE, ainsi que l'ajout de certaines questions intermédiaires ou indications afin de faciliter la résolution des questions les plus difficiles.

Il n'y a que 19 questions, dont certaines sont (très) faciles. Vous devez vous efforcer de chercher tout le sujet et de rédiger avec soin. De plus, certaines questions faciles se trouvent après des questions beaucoup plus difficiles et peuvent être traitées indépendamment. Afin de vous aider dans l'ordre de traitement des questions, chaque question est étiquetée par une estimation de sa difficulté (de ⋆ à ⋆⋆⋆)

Toutes les questions de programmation doivent être traitées en OCaml.
On donne en annexe la documentation de certaines fonctions des modules `List` et `Array` ; celles-ci peuvent être utilisées librement dans toutes les questions.

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.
Tout résultat établi dans une question ou sous-question peut être admis afin de répondre à une autre question ou sous-question.
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 `E`, soit un nœud constitué d'un sous-arbre gauche g, d'un entier x et d'un sous-arbre droit d, noté `N(g, x, d)`. La *taille* d'un arbre t, notée $|t|$, est définie récursivement de la manière suivante :

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

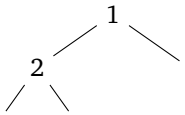
La *hauteur* d'un arbre t, notée $h(t)$, est définie récursivement de la manière suivante :

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

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

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

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 `N(N(E, 2, E), 1, E)` pourra être dessiné sous la forme :

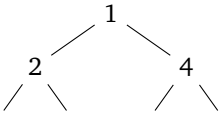


On se donne le type `arbre` suivant pour représenter les arbres binaires en OCaml :

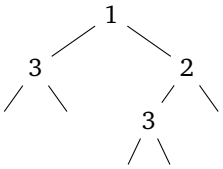
```
type arbre = E | N of arbre * int * arbre
```

Partie I. Structure d'arbre croissant

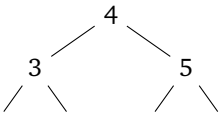
On dit qu'un arbre t est un *arbre croissant* si, soit $t = \mathtt{E}$, soit $t = \mathtt{N}(g, x, d)$ où g et d sont eux-mêmes deux arbres croissants et x est inférieur ou égal à tous les éléments de g et d. Ainsi les arbres :



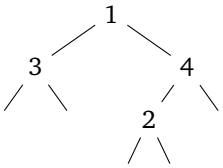
et



sont des arbres croissants mais les arbres :



et



n'en sont pas.

1. ⋆ Écrire une fonction :
   `minimum : arbre -> int`
   qui prend en argument un arbre croissant t, **en supposant** $t \neq \mathtt{E}$, et renvoie son plus petit élément.
2. ⋆ Écrire une fonction :
   `est_un_arbre_croissant : arbre -> bool`
   qui prend en argument un arbre t et détermine s'il a la structure d'arbre croissant. On garantira une complexité $O(|t|)$, qu'on justifiera *succinctement*.
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).
   Indication. Procédez par récurrence forte sur la taille des arbres croissants.

Partie II. Opérations sur les arbres croissants

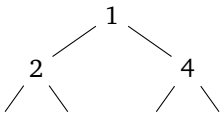
L'opération de *fusion* de deux arbres croissants t_1 et t_2 , notée $\mathtt{fusion}(t_1, t_2)$, est définie par récurrence de la manière suivante :

$$
\begin{aligned}
\mathtt{fusion}(t_1, \mathtt{E}) &= t_1 \\
\mathtt{fusion}(\mathtt{E}, t_2) &= t_2 \\
\mathtt{fusion}(\mathtt{N}(g_1, x_1, d_1), \mathtt{N}(g_2, x_2, d_2)) &= \mathtt{N}(\mathtt{fusion}(d_1, \mathtt{N}(g_2, x_2, d_2)), x_1, g_1) \qquad \text{si } x1 \leq x2 \\
\mathtt{fusion}(\mathtt{N}(g_1, x_1, d_1), \mathtt{N}(g_2, x_2, d_2)) &= \mathtt{N}(\mathtt{fusion}(d_2, \mathtt{N}(g_1, x_1, d_1)), x_2, g_2) \qquad \text{sinon}
\end{aligned}
$$

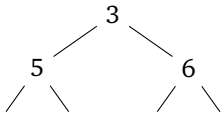
Note importante : dans la troisième (resp. la quatrième) ligne de cette définition, on a sciemment échangé les sous-arbres g_1 et d_1 (resp. g_2 et d_2). Dans les parties III et IV de ce problème apparaîtront les avantages de la fusion telle que réalisée ci-dessus (d'autres façons de réaliser la fusion n'auraient pas nécessairement de telles propriétés).
Dans toute la suite du sujet, on considérera que cette fonction `fusion` est implémentée récursivement en suivant « naturellement » cette définition, par exemple comme ci-dessous :

```
let rec fusion (t1, t2) = match (t1, t2) with
    | t, E | E, t -> t
    | N (g1, x1, d1), N (g2, x2, d2)
          -> if x1 <= x2
             then N ((fusion (d1, t2)), x1, g1)
             else N ((fusion (d2, t1)), x2, g2)
```

4. ⋆ Donner le résultat de la fusion des arbres croissants :



et



5. ⋆⋆⋆ Soit t le résultat de la fusion de deux arbres croissants t_1 et t_2. Montrer que :

- **a.** t possède la structure d'arbre croissant et que,
- **b.** pour tout entier x, $occ(x,t) = occ(x,t_1) + occ(x,t_2)$.

Indication. Procédez à nouveau par récurrence forte sur la taille des arbres croissants.

6. ⋆ Déduire de la fonction `fusion` une fonction :

`ajoute: int -> arbre -> arbre`

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 $y \neq x$.
Remarque. Le sens n'a pas d'importance, sauf pour la question 8...

7. ⋆ Déduire de la fonction `fusion` une fonction :

`supprime_minimum: arbre -> arbre`

qui prend en argument un arbre croissant t, en supposant $t \neq \mathtt{E}$, 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 $y \neq m$.

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

`ajouts_successifs: int array -> arbre`

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

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 $n/2$. On justifiera *succinctement*.

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.
Indication. On *pourra* structurer la réponse à l'aide des sous-questions suivantes, où l'on continue de noter $t_1, \ldots, t_n$ les arbres obtenus lors des ajouts successifs de $1, 2, \ldots, n$ (ou plus généralement, d'entiers strictement croissants).

- **a.** Montrer que l'étiquette de x_i dans t_{i+1} se trouve sur une feuille (un nœud de la forme $\mathtt{N}(\mathtt{E}, x_i, \mathtt{E})$).
- **b.** Montrer que pour tout $1 \leq 2i+1$, t_{2i+1} est de la forme $\mathtt{N}(g_i, 1, d_i)$ avec g_i et d_i deux arbres identiques aux valeurs de leurs étiquettes près.
- **c.** Montrer que pour tout $1 \leq i$, les nœuds de t_i avec au moins un enfant égal à $\mathtt{E}$ (les nœuds « non complets ») occupent au plus deux niveaux de profondeurs, qui sont consécutifs. (Autrement dit, les arbres t_i sont complets, sauf éventuellement à leur dernier niveau.)
- **d.** Conclure.

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.

11. ⋆ ⋆ Écrire une fonction :

```
potentiel: arbre -> int
```

qui prend en argument un arbre t et renvoie $\Phi(t)$, **tout en garantissant une complexité** $O(|t|)$. On justifiera *succinctement* la complexité obtenue.

Remarque. On valorisera partiellement à la correction une fonction correcte qui n'atteint pas la complexité cible, et ce d'autant plus si la complexité de la fonction écrite est correctement identifiée.

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 $C(t,\mathtt{E}) = C(\mathtt{E},t) = 0$.

12. ⋆ ⋆ ⋆ Soient t_1 et t_2 deux arbres croissants et t le résultat de $\mathtt{fusion}(t_1,t_2)$. On veut montrer que :

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

a. Montrez que pour $t_1 = \mathtt{N}(g_1,x_1,d_1)$ et $t_2 = \mathtt{N}(g_2,x_2,d_2)$ non vides, on peut sans perte de généralité considérer que $x_1 \leq x_2$, et l'on a alors $C(t_1,t_2) = 1 + C(d_1,t_2)$.

b. Pour $t_1 \neq \mathtt{E}$, donnez des égalités ou inégalités reliant $\Phi(t_1)$ et $\Phi(g_1)$ & $\Phi(d_1)$; $\log|t_1|$ et $\log|d_1|$, quand :

— $|g_1| < |d_1|$
— $|g_1| \geq |d_1|$

c. Conclure.

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 $O(n\log(n))$.

Indication. Ne pas hésiter à écrire la somme !

14. ⋆ ⋆ ⋆ Montrer que, dans la construction de la question précédente, pour n impair et des valeurs x_i satisfaisant :

— $x_{2i} = x_{2i+1}$ (quand $2i+1 < n-1$);
— $x_i < x_{i-2}$ (quand $i < n$ et $i-2 \geq 0$) ;
— $x_{n-1} > x_i$ pour tout $i < n-1$;

alors l'une des opérations de fusion a un coût au moins égal à $n/2$.
(Un exemple de telles valeurs x_i pour $n = 7$ est : $2,2,1,1,0,0,3$.)
Justifier alors la notion de *complexité amortie* logarithmique pour la fusion de deux arbres croissants.

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 $O(n\log(n))$.

Indication. Procédez comme pour la question 13.

Partie IV. Applications

16. ⋆ En utilisant la structure d'arbre croissant, écrire une fonction :

```
tri : int array -> unit
```

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.

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 :

— $t_0^j = \mathtt{N}(\mathtt{E}, x_j, \mathtt{E})$ pour $0 \leq j < 2^k$,
— $t_{i+1}^j = \mathtt{fusion}(t_i^{2j}, t_i^{2j+1})$ pour $0 \leq i < k$ et $0 \leq j < 2^{k-i-1}$.

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

Indication. Procédez comme pour les questions 13 & 15. On pourra utiliser le fait que $\sum_{i=1}^{\infty} i/2^i$ converge (est majorée par une constante), et utiliser l'approximation $|t_i^*| = 2^i$ (on ne commet ainsi qu'une erreur constante pour chaque arbre de la construction, ce qui ne change pas le résultat).

18. ⋆⋆ En déduire une fonction :

```
construire : int array -> arbre
```

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 $O(n)$.

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.

Annexe : Aide à la programmation en OCaml

Opérations sur les listes : Le module `List` offre les fonctions suivantes :

— `List.length : 'a list -> int`
Return the length (number of elements) of the given list.
— `List.mem : 'a -> 'a list -> bool`
`List.mem a set` is true if and only if `a` is equal to an element of `set`.
— `List.exists : ('a -> bool) -> 'a list -> bool`
`List.exists f [a1; ...; an]` checks if at least one element of the list satisfies the predicate `f`. That is, it returns `(f a1) || (f a2) || ... || (f an)` for a non-empty list and `false` if the list is empty.
— `List.for_all : ('a -> bool) -> 'a list -> bool`
`List.for_all f [a1; ...; an]` checks if all elements of the list satisfy the predicate `f`. That is, it returns `(f a1) && (f a2) && ... && (f an)` for a non-empty list and `true` if the list is empty.
— `List.filter : ('a -> bool) -> 'a list -> 'a list`
`List.filter f l` returns all the elements of the list `l` that satisfy the predicate `f`. The order of the elements in the input list is preserved.
— `List.map : ('a -> 'b) -> 'a list -> 'b list`
`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.

Opérations sur les tableaux : Le module `Array` offre les fonctions suivantes :

— `Array.length : 'a array -> int`
Return the length (number of elements) of the given array.
— `Array.make : int -> 'a -> 'a array`
`Array.make n x` returns a fresh array of length n, initialized with x. All the elements of this new array are initially physically equal to x (in the sense of the == predicate). Consequently, if x is mutable, it is shared among all elements of the array, and modifying x through one of the array entries will modify all other entries at the same time.
— `Array.make_matrix : int -> int -> 'a -> 'a array array`
`Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension *dimx* and second dimension *dimy*. All the elements of this new matrix are initially physically equal to e. The element (x, y) of a matrix m is accessed with the notation `m.(x).(y)`.
— `Array.init : int -> (int -> 'a) -> 'a array`
`Array.init n f` returns a fresh array of length n, with element number i initialized to the result of $f(i)$. In other terms, `init n f` tabulates the results of f applied to the integers 0 to $n-1$.

— `Array.copy : 'a array -> 'a array`
`Array.copy a` returns a copy of a, that is, a fresh array containing the same elements as a.
— `Array.mem : 'a -> 'a array -> bool`
`Array.mem a l` is true if and only if a is structurally equal to an element of l (i.e. there is an x in l such that `compare a x = 0`).
— `Array.for_all : ('a -> bool) -> 'a array -> bool`
`Array.for_all f [|a1; ...; an|]` checks if all elements of the array satisfy the predicate f. That is, it returns `(f a1) && (f a2) && ... && (f an)`.
— `Array.exists : ('a -> bool) -> 'a array -> bool`
`Array.exists f [|a1; ...; an|]` checks if at least one element of the array satisfies the predicate f. That is, it returns `(f a1) || (f a2) || ... || (f an)`.
— `Array.map : ('a -> 'b) -> 'a array -> 'b array`
`Array.map f a` applies function f to all the elements of a, and builds an array with the results returned by f : `[| f a.(0); f a.(1); ...; f a.(length a - 1) |]`.
— `Array.iter : ('a -> unit) -> 'a array -> unit`
`Array.iter f a` applies function f in turn to all the elements of a. It is equivalent to `f a.(0); f a.(1); ...; f a.(length a - 1); ()`.

D'après https://ocaml.org/manual/5.2/api/index.html