Devoir surveillé #4 (avec solutions)

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 :

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

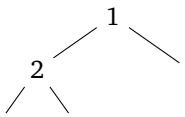
La *hauteur* d'un arbre t, notée $h(t)$, est définie récursivement de la manière suivante :

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

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 :

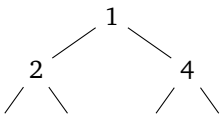
```
type arbre = E | N of arbre * int * arbre
```

Avec ce type, les définitions ci-dessus peuvent se traduire en fonctions OCaml comme :

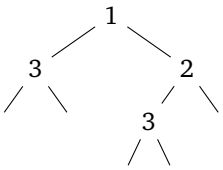
```
let rec size = function E -> 1 | N (l,_,r) -> 1 + (size l) + (size r)

let rec height = function E -> 0 | N (l,_,r) -> 1 + max (height l) (height r)

let rec occ y = function
    | E -> 0
    | N (l,x,r) -> let p = if x = y then 1 else 0 in p + (occ y l) + (occ y r)
```

Partie I. Structure d'arbre croissant

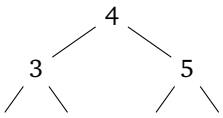
On dit qu'un arbre t est un *arbre croissant* si, soit $t = \mathtt{E}$, soit $t = \mathtt{N}(g, x, d)$ où g et d sont eux-mêmes deux arbres croissants et x est inférieur ou égal à tous les éléments de g et d. Ainsi les arbres :



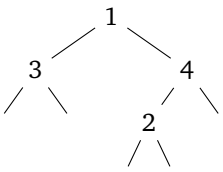
et



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.

   C'est immédiat, en exploitant le fait que dans un arbre croissant l'élément recherché est nécessairement à la racine.
   ```
   let minimum = function E -> assert false | N (_,x,_) -> x
   ```

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*.

   Il faut prendre garde à ne pas appeler la fonction *minimum* sur un arbre vide ; il suffit sinon d'appliquer directement la définition, en observant qu'il est inutile de comparer la racine à *tous* les éléments des enfants, si ceux-ci sont des arbres croissants (puisqu'alors leurs minima sont en leurs propres racines). On propose :

```
let est_un_arbre_croissant t =
    let rec ac m = function
        | E -> true
        | N (l, x, r) -> (x >= m) && (ac x l) && (ac x r)
    in
    match t with
    | E -> true
    | N (l, x, r) -> (ac x l) && (ac x r)
```

On atteint bien la complexité cible, puisque chaque nœud est visité au plus une fois, avec un traitement marginal de coût constant (*viz.* une comparaison).

3. ⋆ ⋆ Montrer qu'il y a exactement $n!$ arbres croissants possédant n nœuds étiquetés par les entiers $1, \dots, n$ (chaque nœud étant étiqueté par un entier distinct).
Indication. Procédez par récurrence forte sur la taille des arbres croissants.

On procède par récurrence forte sur n.

— Cas de base $n \leq 1$: il existe exactement un arbre croissant de un nœud étiqueté par $\{1\}$ (*viz.* $\mathtt{N}(\mathtt{E}, 1, \mathtt{E})$) et un arbre de zéro nœuds étiqueté par $\emptyset$ (*viz.* $\mathtt{E}$).

— Cas récursif : on suppose la propriété vraie aux rangs $\leq n$, et l'on montre qu'elle reste vraie au rang $n+1$.
Par la définition d'un arbre croissant, son minimum est nécessairement à la racine. Un arbre croissant d'éléments $\{1, \dots, n+1\}$ est donc de la forme $\mathtt{N}(g, 1, d)$ avec g et d eux-mêmes des arbres croissants d'éléments $\mathcal{G}$ et $\mathcal{D}$ respectivement, avec $\mathcal{G} \cup \mathcal{D} = \{2, \dots, n+1\}$.
Pour une partition $\mathcal{G}, \mathcal{D}$ donnée, on a par hypothèse de récurrence que le nombre d'arbres croissants à gauche possible est $|\mathcal{G}|!$: en effet, $|G| \leq n$, et les arbres croissants d'étiquettes dans $\mathcal{G}$ sont en bijection avec les arbres croissants d'étiquettes dans $\{1, \dots, |\mathcal{G}|\}$: il suffit de renuméroter les étiquettes par ordre croissant avec les entiers consécutifs depuis 1 (ce qui, étant donné $\mathcal{G}$, est inversible). Il en va de même pour le nombre d'arbres croissants à droite.
Il suffit maintenant de sommer le nombre de partitions possibles pour un cardinal de $\mathcal{G}$ fixé, pour chaque cardinal possible, et l'on obtient :

$$\sum_{i=0}^{n} \binom{n}{i} \times i! \times (n-i)! = \sum_{i=0}^{n} n! = (n+1)!$$

Partie II. Opérations sur les arbres croissants

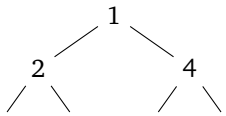
L'opération de *fusion* de deux arbres croissants t_1 et t_2 , notée $\mathtt{fusion}(t_1, t_2)$, est définie par récurrence de la manière suivante :

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

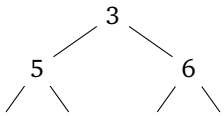
Note importante : dans la troisième (resp. la quatrième) ligne de cette définition, on a sciemment échangé les sous-arbres g_1 et d_1 (resp. g_2 et d_2). Dans les parties III et IV de ce problème apparaîtront les avantages de la fusion telle que réalisée ci-dessus (d'autres façons de réaliser la fusion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

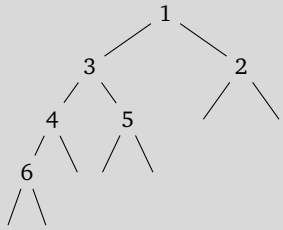
4. ⋆ Donner le résultat de la fusion des arbres croissants :



et



On obtient :



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.

On procède (essentiellement) par récurrence forte sur la somme des tailles des arguments (soit sur $|t_1| + |t_2|$).

a. — Cas de base : le cas de base correspond à t_1, t_2 arbres vides et une somme des tailles égale à 2. Dans ce cas le résultat de la fusion est l'arbre vide, qui est un arbre croissant. De plus, dans n'importe quel cas où l'un des arguments est `E`, le résultat de la fusion est l'autre argument qui est croissant par hypothèse.

— Cas récursif : On suppose la propriété vraie pour tous les arguments non vides dont la somme des tailles est $\leq n$, et l'on montre la conservation pour $n+1$.
On suppose également $x_1 \leq x_2$ (le second cas est symétrique).
Les deux arguments t_1 et t_2 sont de la forme $\texttt{N}(g_1, x_1, d_1)$ et $\texttt{N}(g_2, x_2, d_2)$, avec t_1, g_1, d_1, t_2, g_2, d_2 tous croissants par hypothèse. De plus, par définition de la taille (et $|t_1| + |t_2| = n+1$), tous ces arbres (y compris t_1 et t_2) sont de taille $\leq n$.
Par hypothèse de récurrence forte, $\texttt{fusion}(d_1, t_2)$ est un arbre croissant, dont (par l'hypothèse $x_1 \leq x_2$ et le fait que t_1 & t_2 sont croissants) les éléments sont plus grands que x_1. Le résultat :

$$t = \texttt{N}(\texttt{fusion}(d_1, t_2), x_1, g_1)$$

de la fusion de t_1 et t_2 est donc bien un arbre croissant, et la propriété est conservée.

b. — Cas de base : comme pour **a.**, en utilisant cette fois le fait que $occ(x, \texttt{E})$ vaut 0 pour tout x.

— Cas récursif :
On suppose à nouveau la propriété vraie pour tous les arguments non vides dont la somme des tailles est $\leq n$, et l'on montre la conservation pour $n+1$.
On suppose également toujours $x_1 \leq x_2$ (le second cas est symétrique).
En réutilisant les mêmes notations qu'en **a.**, on a toujours t_1, g_1, d_1, t_2, g_2, d_2 de taille $\leq n$. On note alors $t'_1 = \texttt{fusion}(d_1, t_2)$, et l'on a $t = \texttt{N}(t'_1, x_1, g_1)$.
Par définition de occ, on a $occ(x,t) = occ(t'_1) + occ(g_1) + [x = x_1]$, où $[x = x_1]$ vaut 1 si $x = x_1$, et 0 sinon.
Par hypothèse de récurrence forte, $occ(x, t'_1) = occ(x, d_1) + occ(x, t_2)$ pour tout x.
On a donc (pour tout x) :

$$occ(x,t) = occ(x,t_2) + occ(x,d_1) + occ(g_1) + [x = x_1]$$

Or :

$$occ(x,t_1) = occ(x,d_1) + occ(g_1) + [x = x_1]$$

ce qui permet de conclure.

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…

Il suffit de correctement appeler *fusion*.

```
let ajoute x t = fusion (t, (N (E, x, E)))
```

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 $y \neq m$.

Ditto.

```
let supprime_minimum = function
    | E -> assert false
    | N (l, _, r) -> fusion (l,r)
```

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.

On peut procéder de plusieurs façons : récursivement, avec une boucle `for`, ou encore en utilisant un itérateur (plusieurs possibilités). Ceci donne par exemple :

```
let ajouts_successifs v =
    let t = ref E in
    Array.iter (fun x -> t := ajoute x !t) v ; !t

let ajouts_successifs' v =
    let t = ref E in
    let n = Array.length v in
    for i = 0 to (n - 1) do
        t := ajoute v.(i) !t
    done ;
    !t

let ajouts_successifs'' v =
    let n = Array.length v in
    let rec loop i t =
        if i = n then t
        else loop (i + 1) (ajoute v.(i) t)
    in loop 0 E
```

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*.

Une suite de valeurs strictement décroissantes convient : si $x_0 > x_1 > \cdots > x_{n-1}$, l'arbre construit par *ajouts_successifs* aura une hauteur $n > n/2$.
On justifie brièvement par récurrence sur n : le cas de base $n = 1$ est évident (pour la définition de *hauteur* du sujet), et par définition de *fusion*, l'ajout d'un élément x_n plus *petit* que tous les autres à un arbre t de hauteur n produira un arbre $\mathtt{N}(t, x_n, \mathtt{E})$ de hauteur $n+1$ (l'arbre obtenu est un peigne à gauche). (On peut par ailleurs remarquer que chacun de ces ajouts se fait en temps constant.)

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.

a. On a que t_i est obtenu par ajout de x_i à t_{i-1}, c'est à dire par fusion de t_{i-1} et $\mathtt{N}(\mathtt{E}, x_i, \mathtt{E})$ (on notera t^i ce dernier arbre ci-dessous). On va prouver la propriété plus généralement pour n'importe quel arbre t dont les éléments sont tous strictement plus petits que x_i, ce qui est bien le cas de t_{i-1}.
On procède par récurrence forte sur la taille de t. Le cas de base $|t| = 1$ correspond à $t = \mathtt{E}$ et est évident. Dans le cas récursif l'on a $t = \mathtt{N}(g, x, d)$ avec $|d| < |t|$. Puisque par hypothèse $x_i > x$, on a $\mathtt{fusion}(t, t^i) = \mathtt{N}(\mathtt{fusion}(d, t^i), x, g)$. Par hypothèse de récurrence forte,, x_i est une feuille de $\mathtt{fusion}(d, t^i)$, et donc de $\mathtt{fusion}(t, t^i)$.

b. On procède par récurrence sur $2i+1$. Le cas de base $2i+1 = 1$ est vérifié (avec $t_1 = \mathtt{N}(\mathtt{E}, 1, \mathtt{E})$).
On suppose la propriété vraie pour l'entier impair $2i+1$, et l'on montre qu'elle est préservée pour le prochain entier impair $2i+3$. Par définition de l'ajout et le fait que $x_{2i+3} > x_{2i+2} > 1$, on a :

— $t_{2i+2} = \mathtt{N}(\mathtt{fusion}(d_i, \mathtt{N}(\mathtt{E}, x_{2i+2}, \mathtt{E})), 1, g_i)$
— $t_{2i+3} = \mathtt{N}(\mathtt{fusion}(g_i, \mathtt{N}(\mathtt{E}, x_{2i+3}, \mathtt{E})), 1, \mathtt{fusion}(d_i, \mathtt{N}(\mathtt{E}, x_{2i+2}, \mathtt{E})))$

Par hypothèse de récurrence, g_i et d_i sont identiques à leurs étiquettes près, toutes inférieures à x_{2i+2} et x_{2i+3}. Il en résulte (par une récurrence forte identique à **a.**) que $\mathtt{fusion}(d_i, \mathtt{N}(\mathtt{E}, x_{2i+2}, \mathtt{E}))$ et $\mathtt{fusion}(g_i, \mathtt{N}(\mathtt{E}, x_{2i+3}, \mathtt{E}))$ sont également identiques à leurs étiquettes près, ce qui conclut la récurrence.
La propriété reste vraie en général pour toute suite d'entiers strictement croissants (à la valeur de l'étiquette à la racine près).

c. On procède par récurrence forte sur le nombre d'entiers ajoutés, en montrant la propriété générale pour l'ajout de n'importe quelle suite d'entiers strictement croissants. Le cas de base $i = 1$ est vérifié pour t_1.
On suppose la propriété vraie pour i, et l'on montre qu'elle est préservée pour $i+1$. On note g_i et d_i les sous-arbres gauche et droit de $t_i = \mathtt{N}(g_i, 1, d_i)$, et g_{i+1} et g_i ceux de $t_{i+1} = \mathtt{N}(g_{i+1}, 1, g_i)$ (nécessairement de cette forme (à la valeur de la racine près) par hypothèse sur la croissance des x_i's).
On considère deux cas en fonction de la parité de i :

— si i est pair, t_{i+1} satisfait **b.** et g_{i+1} et g_i sont identiques à leurs étiquettes près. Par hypothèse de récurrence t_i est complet, sauf éventuellement à son dernier niveau ; c'est aussi vrai pour g_i par hypothèse de récurrence (qui est un arbre obtenu par ajout successif de $< i$ entiers strictement croissants), et donc pour t_{i+1}.
— si i est impair, t_i satisfait **b.**, et g_i et d_i sont donc identiques à étiquettes près. Par hypothèse de récurrence, les nœuds de t_i avec au moins un enfant égal à $\mathtt{E}$ sont situés à au plus deux niveaux de profondeur, que l'on note $k+1$ et éventuellement $k+2$; par la structure de t_i, ces mêmes nœuds sont situés à au plus deux niveaux de profondeur k et éventuellement $k+1$ de g_i et d_i.
Par **a.** et la définition de l'ajout, l'étiquette de x_{i+1} est une feuille dans g_{i+1}, et par hypothèse de récurrence, les nœuds non complets de g_{i+1} occupent au plus deux niveaux de profondeur consécutifs.
On distingue alors deux cas en fonction des niveaux occupés par les nœuds non complets de g_i :

— $\{k\}$: x_{i+1} est ajoutée comme une feuille, donc nécessairement au niveau $k+1$: les nœuds non complets de g_{i+1} occupent au plus les niveaux k et $k+1$;
— $\{k, k+1\}$: x_{i+1} est ajoutée comme une feuille, donc à un niveau $\in \{k+1, k+2\}$. Si elle l'était au niveau $k+2$, cela ne modifierait aucun nœud non complet de niveau k, et les niveaux de g_{i+1} occupés par des nœuds non complets seraient alors $\{k, k+1, k+2\}$, ce qui est absurde. Donc x_{i+1} ne peut qu'être ajoutée au niveau $k+1$, et les niveaux non complets de g_{i+1} sont $\subseteq \{k, k+1\}$.

Dans les deux cas la propriété est préservée pour t_{i+1}.

d. Par construction t_n possède n nœuds ; par **c.**, c'est un arbre complet sauf éventuellement à son dernier niveau. Soit h sa hauteur, t_n a donc strictement plus de de nœuds qu'un arbre complet de hauteur $h-1$, et au plus autant qu'un arbre complet de hauteur h, ce qui donne la suite d'encadrements (attention à la définition de hauteur du sujet !) :

$$2^{h-1} - 1 < n \leq 2^h - 1$$
$$2^{h-1} < n + 1 \leq 2^h$$
$$h - 1 < \log(n+1) \leq h$$

d'où $h = \lceil \log(n+1) \rceil$.

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.

La fonction demandée ne comporte pas de difficultés particulières : il suffit d'appliquer la définition du potentiel. Cependant, pour obtenir le coût cible, on ne peut pas se permettre le recalculer la taille des enfants en permanence : ceci engendrerait un coût égal à la somme des tailles de tous les sous-arbres, qui n'est en général pas linéaire en $|t|$ (ce dont on se convainc aisément en considérant par exemple un arbre peigne). Une solution suffisante consiste alors à utiliser une fonction intermédiaire qui calcule conjointement potentiel et hauteur :

```
let potentiel t =
    let rec pot = function
        | E -> (1, 0) (* taille, #nœuds lourds *)
        | N (l, _, r) ->
                let sl, hnl = pot l in
                let sr, hnr = pot r in
                (1 + sl + sr,
                 hnl + hnr + (if sl < sr then 1 else 0))
    in snd (pot t)
```

(l'utilisation de `snd` est ici bien pratique, mais pas essentielle).
Cette fonction atteint bien le coût cible, puisque chaque nœud n'est visité qu'une fois, avec un traitement marginal de coût constant (*viz.* quelques additions, une comparaison, un test).

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.

a. On peut supposer sans perte de généralité que $x_1 \leq x_2$, puisque l'analyse est complètement symétrique dans le cas inverse. Par définition, on a alors $\mathtt{fusion}(t_1, t_2)$ égal à $\mathtt{N}(\mathtt{fusion}(d_1, t_2), x_1, g_1)$, et le calcul de $\mathtt{fusion}(t_1, t_2)$ effectue donc un appel récursif à $\mathtt{fusion}(d_1, t_2)$, plus tous les appels récursifs effectués par le calcul de $\mathtt{fusion}(d_1, t_2)$, ce qui (par définition de C) donne bien $C(t_1, t_2) = 1 + C(d_1, t_2)$.

b. Cas $|g_1| < |d_1|$:

— Par définition de Φ (notamment son calcul récursif comme à la question précédente), $\Phi(t_1) = 1 + \Phi(g_1) + \Phi(d_1)$.
— Par définition de la taille et par hypothèse, on a $|t_1| = |g_1| + |d_1| + 1 \leq 2|d_1|$, d'où $\log|t_1| \leq \log|d_1| + 1$.

Cas $|g_1| \geq |d_1|$:

— On a $\Phi(t_1) = \Phi(g_1) + \Phi(d_1)$.
— On a $|t_1| = |g_1| + |d_1| + 1 > 2|d_1|$, d'où $\log|t_1| > \log|d_1| + 1$,

c. On montre le résultat (essentiellement) par récurrence forte sur $|t_1| + |t_2|$.
On prend comme cas de base t_1 ou t_2 de taille 1, soit l'un des deux égal à $\mathtt{E}$; on a dans ce cas $C(t_1, t_2) = 0$.
Supposons sans perte de généralité que $t_2 = \mathtt{E}$; on a alors :

$$\begin{aligned} &\Phi(t_1) + \Phi(t_2) - \Phi(t) + 2(\log|t_1| + \log|t_2|) = \\ &\Phi(t_1) + \Phi(\mathtt{E}) - \Phi(t_1) + 2(\log|t_1| + \log|t_2|) = && \text{par définition de } t_2 \\ &\qquad 2(\log|t_1| + \log|t_2|) \geq 0 && \text{puisque } \Phi(\mathtt{E}) = 0 \end{aligned}$$

On traite le cas récursif en supposant sans perte de généralité que $x_1 \leq x_2$, et séparément en fonction de si $|g_1| < |d_1|$ ou non.
Cas $|g_1| < |d_1|$:
On note t' le résultat de la fusion de d_1 et t_2, et l'on remarque que $|t'| \geq |d_1| > |g_1|$ par hypothèse, et donc $\Phi(t) = \Phi(t') + \Phi(g_1)$ (ou : $\Phi(t') = \Phi(t) - \Phi(g_1)$) ; on a alors :

$$\begin{aligned} C(t_1,t_2) &= 1 + C(d_1,t_2) && \text{par } \textbf{a.} \\ &\leq \Phi(d_1) + \Phi(t_2) - \Phi(t') + 2(\log|d_1| + \log|t_2|) + 1 && \text{par hypothèse de récurrence forte} \\ &= (\Phi(t_1) - \Phi(g_1) - 1) + \Phi(t_2) - \Phi(t') + 2(\log|d_1| + \log|t_2|) + 1 && \text{par } \textbf{b.} \\ &= \Phi(t_1) + \Phi(t_2) - \Phi(t) + 2(\log|d_1| + \log|t_2|) && \text{remarque ci-dessus} \\ &\leq \Phi(t_1) + \Phi(t_2) - \Phi(t) + 2(\log|t_1| + \log|t_2|) && |d_1| \leq |t_1| \end{aligned}$$

Cas $|g_1| \geq |d_1|$:
Avec les mêmes notations que ci-dessus, on remarque que l'on a $\Phi(t) \leq 1 + \Phi(g_1) + \Phi(t')$ (ou : $\Phi(t) - \Phi(g_1) - 1 \leq \Phi(t')$) ; on a alors :

$$\begin{aligned} C(t_1,t_2) &= 1 + C(d_1,t_2) && \text{par } \textbf{a.} \\ &\leq \Phi(d_1) + \Phi(t_2) - \Phi(t') + 2(\log|d_1| + \log|t_2|) + 1 && \text{par hypothèse de récurrence forte} \\ &= (\Phi(t_1) - \Phi(g_1)) + \Phi(t_2) - \Phi(t') + 2(\log|d_1| + \log|t_2|) + 1 && \text{par } \textbf{b.} \\ &\leq \Phi(t_1) + \Phi(t_2) - \Phi(t) + 2(\log|d_1| + \log|t_2|) + 2 && \text{remarque ci-dessus} \\ &\leq \Phi(t_1) + \Phi(t_2) - \Phi(t) + 2(\log|t_1| + \log|t_2|) && \text{par } \textbf{b.} \end{aligned}$$

13. $\star\star$ 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 !

La construction peut s'obtenir par un simple appel à *ajouts_successifs*, qui effectue n appels à *ajoute*, dont chacun effectue un appel à *fusion*. En explicitant ces derniers appels, la construction peut se faire pour un coût donné par :

$$An + \sum_{i=0}^{n-1} C(t_i, \mathtt{N}(\mathtt{E}, x_i, \mathtt{E}))$$

avec A une constante.
Par abus de notation, on utilise x_i pour $\mathtt{N}(\mathtt{E}, x_i, \mathtt{E})$, et l'on remarque que $|x_i| = 3$ et $\Phi(x_i) = 0$ pour tout i. On utilise alors le résultat de la question précédente pour majorer le coût par :

$$
\begin{aligned}
& An + \sum_{i=0}^{n-1} \Phi(t_i) + \Phi(x_i) - \Phi(t_{i+1}) + 2(\log|t_i| + \log|x_i|) && \text{question 12} \\
= \quad & An + \sum_{i=0}^{n-1} \Phi(t_i) - \Phi(t_{i+1}) + 2(\log|t_i| + \log 3) && \text{remarque ci-dessus} \\
= \quad & An + \Phi(t_0) - \Phi(t_n) + \sum_{i=0}^{n-1} 2(\log|t_i| + \log 3) && \text{télescopage} \\
\leq \quad & Bn \log n && B \text{ constante}
\end{aligned}
$$

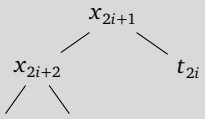
où la dernière inégalité vient du fait que $\Phi(t_0) = 0 \leq n$, $\Phi(t_n) \geq 0$, et $|t_i|$ est un $O(n)$ pour tout i.

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.

On commence par montrer que dans la construction ci-dessus, t_{2i+2} est égal à $\mathtt{N}(\mathtt{N}(\mathtt{E}, x_{2i+2}, \mathtt{E}), x_{2i+1}, t_{2i})$, c'est à dire de la forme :



(Ces arbres sont donc quasiment des peignes à droite.)
Ceci se prouve directement par calcul pour toute valeur $2i+2 < n-1$:

— $t_{2i+1} = \mathtt{N}(t_{2i}, x_{2i+1}, \mathtt{E})$ par le fait que $x_{2i+1} < x_{2i} = x_{2i-1}$;
— $t_{2i+2} = \mathtt{N}(\mathtt{N}(\mathtt{E}, x_{2i+2}, \mathtt{E}), x_{2i+1}, t_{2i+1})$ par le fait que $x_{2i+1} \leq x_{2i+2}$.

Bien que ce ne soit pas essentiel pour la suite, on peut remarquer que toutes les constructions des $t_{i<n}$ sont de coût constant, mais que le potentiel est lui croissant : $\Phi(t_{2i+2}) = i$.
On montre maintenant par récurrence qu'ajouter x_{n-1} à t_{2i} ($2i < n-1$) coûte au moins i appels récursifs à *fusion*.

— C'est trivialement vrai pour le cas de base $i = 0$
— On suppose la propriété vraie pour $2i$. Par l'analyse ci-dessus, on a (pour $2i+2 < n-1$) que $t_{2i+2} = \mathtt{N}(\mathtt{N}(\mathtt{E}, x_{2i+2}, \mathtt{E}), x_{2i+1}, t_{2i+1})$. Par hypothèse $x_{n-1} > x_{2i+1}$ et donc $\mathtt{fusion}(t_{2i+2}, \mathtt{N}(\mathtt{E}, x_{n-1}, \mathtt{E}))$ est égal (et est calculé comme) :

$$\mathtt{N}(\mathtt{fusion}(t_{2i}, \mathtt{N}(\mathtt{E}, x_{n-1}, \mathtt{E})), x_{2i+1}, \mathtt{N}(\mathtt{E}, x_{2i+2}, \mathtt{E})),$$

ce qui coûte un appel récursif plus les appels récursifs effectués par $\mathtt{fusion}(t_{2i}, \mathtt{N}(\mathtt{E}, x_{n-1}))$, soit au moins $i+1$ appels récursifs au total par hypothèse de récurrence.

La propriété que l'on vient de démontrer est en particulier vraie pour l'ajout de x_{n-1} à t_{n-1}, qui (*via* les appels récursifs à *fusion*) est bien de coût au moins $n/2$.
Bien que ce ne soit pas essentiel pour répondre à la question, on peut observer que t_n a quasiment une structure de peigne à gauche et un potentiel de 0, ce qui est cohérent avec la majoration établie à la question 12 : pour que $C(t_{n-1}, x_{n-1})$ « puisse coûter cher », il faut « consommer » le potentiel de $\Phi(t_{n-1})$, et donc que le potentiel du résultat soit faible.
Pour justifier la *complexité amortie* de la fusion d'arbres croissants, il faut avoir l'intuition (non explicitée par le sujet) que cette analyse porte sur le coût « moyenné » de la fusion dans une suite de fusions partant d'un arbre vide. Le plus cohérent par rapport à la question précédente est alors de spécifiquement considérer la suite des fusions apparaissant lors de la construction de t_n.

La majoration du coût obtenu à cette question reste valide dans le cas un peu plus général où l'on fusionne n arbres de taille $O(n)$ de somme de potentiels un $O(n \log n)$, mais c'est un peu *ad hoc* et probablement pas ce que les concepteurs du sujet avaient à l'esprit.

Dans ce cas, puisque l'ensemble des n fusions a un coût en $O(n \log n)$, chaque fusion n'a un coût *amorti* que de $O((n \log n)/n)$, soit $O(\log n)$. Pour autant, comme le montre l'exemple de cette question, l'une de ces fusions peut avoir un coût **linéaire** en la taille de ses entrées (ici : n). Ceci met en évidence l'utilité d'une analyse amortie (portant sur une *suite* d'opérations) : une approche plus naïve ayant consisté à majorer le coût pire cas d'une fusion et à multiplier par le nombre de fusion n'aurait pas pu conclure à une meilleure borne que $O(n^2)$, tandis que l'analyse amortie montre que ce pire cas ne peut pas « arriver à chaque fois ».

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 $t_{i+1} = \texttt{fusion}(g_i, d_i)$ 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.

On se concentre comme dans la question 13 sur le nombre d'appels récursifs à *fusion*. Par définition, le coût de construction de t_{i+1} est $C(g_i, d_i)$. Par la question 12 et le fait que tous les arbres impliqués dans cette construction sont de taille $O(n)$, celui-ci est majoré par $\Phi(g_i) + \Phi(d_i) - \Phi(t_{i+1}) + B \log n$ (pour une certaine constante B). On a également $\Phi(g_i) + \Phi(d_i) \leq \Phi(t_i)$, d'où l'on déduit $C(g_i, d_i) \leq \Phi(t_i) - \Phi(t_{i+1}) + B \log n$. Finalement, le coût total de la construction peut alors se majorer par :

$$\begin{aligned} & An + \sum_{i=0}^{n-1} \Phi(t_i) - \Phi(t_{i+1}) + B \log n && \text{analyse ci-dessus} \\ = \quad & An + \Phi(t_0) - \Phi(t_n) + \sum_{i=0}^{n-1} B \log n && \text{télescopage} \\ \leq \quad & Cn \log n && C \text{ constante} \end{aligned}$$

où la dernière inégalité vient du fait qu'on a pour tout arbre t que $\Phi(t) \leq |t|$, et donc que $\Phi(t_0)$ est un $O(n)$.

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.

La fonction est aisée à écrire en style impératif, en réutilisant les fonctions déjà écrites. On propose par exemple :

```
let tri v =
    let n = Array.length v in
    let t = ref (ajouts_successifs v) in (* O (n log n) Q.13 *)
    for i = 0 to (n - 1) do (* O (n log n) Q.15 *)
        v.(i) <- minimum !t ;
        t := supprime_minimum !t
    done
```

Le coût se justifie aisément en utilisant les résultats des questions précédentes : *ajouts_successifs* a un coût en $O(n \log n)$ par la question 13, et les n appels à *supprime_minimum* construisent exactement la suite d'arbres décrite à la question 15, encore une fois pour un coût en $O(n \log n)$. Le reste des opérations ont un coût constant, et sont réalisées au plus n fois.

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 $0 \leq j < 2^k$,

— $t_{i+1}^{j} = \texttt{fusion}(t_i^{2j}, t_i^{2j+1})$ pour $0 \le i < k$ et $0 \le j < 2^{k-i-1}$.

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).

On se concentre comme dans les questions 13 & 15 sur le nombre d'appels récursifs à *fusion*.
Le coût de construction des arbres t_0^* du niveau 0 n'occasionne aucun appel récursif et est de coût linéaire en $n = 2^k$.
On note $C(i+1)$ le nombre d'appels récursifs à *fusion* lors de la construction de l'ensemble des arbres t_{i+1}^* du niveau $i+1$. On a alors :

$$\begin{aligned}
C(i+1) &= \sum_{j=0}^{2^{k-i-1}} C(t_i^{2j}, t_i^{2j+1}) && \text{par définition}\\
&\le \sum_{j=0}^{2^{k-i-1}} \Phi(t_i^{2j}) + \Phi(t_i^{2j+1}) - \Phi(t_{i+1}^{j}) + 2(\log|t_i^{2j}| + \log|t_i^{2j+1}|) && \text{question 12}\\
&\le \sum_{j=0}^{2^{k-i-1}} \left[\Phi(t_i^{2j}) + \Phi(t_i^{2j+1}) - \Phi(t_{i+1}^{j})\right] + i \times 2^{k-i+1} && \text{approximation } \log|t_i^*| \approx i
\end{aligned}$$

Le nombre total d'appels récursifs est donc donné par :

$$\begin{aligned}
\sum_{i=0}^{k-1} C(i+1) &\le \sum_{i=0}^{k-1} \left[i \times 2^{k-i+1} + \sum_{j=0}^{2^{k-i-1}} \left[\Phi(t_i^{2j}) + \Phi(t_i^{2j+1})\right] - \sum_{j=0}^{2^{k-i-1}} \Phi(t_{i+1}^{j}) \right]\\
&\le \sum_{i=0}^{k-1} \left[i \times 2^{k-i+1} \right] + \sum_{j=0}^{2^{k-1}} \left[\Phi(t_0^{2j}) + \Phi(t_0^{2j+1})\right] - \Phi(t_k^0) && \text{télescopage}\\
&\le 2^{k+1} \sum_{i=0}^{k-1} \left[i/2^i \right] + \sum_{j=0}^{2^{k-1}} \left[\Phi(t_0^{2j}) + \Phi(t_0^{2j+1})\right] - \Phi(t_k^0)\\
&\le A2^k + 0 - \Phi(t_k^0) && A \text{ constante}\\
&\le B2^k && B \text{ constante}
\end{aligned}$$

Où l'on a utilisé dans les simplifications successives les faits que $\sum_{i=0}^{\infty} i/2^i$ converge, $\Phi(t_0^*) = 0$, et $\Phi(t_k^0) \le |t_k^0| \in O(2^k)$.
Remarque. Cette analyse est similaire (et utilise les mêmes arguments) que celle de la fonction *heapify* du « tri par tas » classique.

18. ⋆⋆ En déduire une fonction :

`construire : int array -> arbre`

qui prend en argument un tableau `a` de taille n, avec $n = 2^k$ et $k \ge 0$, et renvoie un arbre croissant contenant les éléments de `a` en temps $O(n)$.

On suit exactement la construction décrite ci-dessus ; on peut procéder de plusieurs façons, mais en aucun cas supposer l'existence d'une fonction `log2 : int -> int` calculant le logarithme en base 2 (ou que l'on dispose d'un argument correspondant à « k »).
On propose deux variantes : l'une en style impératif qui construit un tableau intermédiaire, l'autre en style fonctionnel.
La complexité ayant été analysée à la question 17, il n'est pas nécessaire de la justifier à nouveau.

```
let construire x =
    let n = ref (Array.length x) in
    let t = Array.init !n (fun j -> N (E, x.(j), E)) in
    while !n > 1 do
        n := !n / 2 ;
```

```
        for i = 0 to (!n - 1) do
            t.(i) <- fusion (t.(2*i), t.(2*i+1))
        done
    done ;
    t.(0)

let construire' x =
    let rec cons n i =
        if n = 1 then N (E, x.(i), E)
        else let tl = cons (n/2) (2*i) in
             let tr = cons (n/2) (2*i+1) in
             fusion (tl,tr)
    in cons (Array.length x) 0
```

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.

Une façon de relâcher la contrainte est de considérer une construction où l'on complète les arbres du premier niveau t_0 par autant d'arbres vides jusqu'à en obtenir un nombre égal à 2^k —la prochaine puissance de 2 supérieure à n— puis à suivre le processus normal de construction. On obtient alors bien un arbre croissant d'étiquettes égales aux x_i's, puisque `E` est un élément neutre pour la fusion. Le coût de cette construction est un $O(2^k)$ par la question 17, qui est aussi un $O(n)$ puisque $2^k < 2n$.
Ceci s'implémente aisément en ajoutant un cas de base à la fonction récursive de *construire'* ci-dessus pour s'évaluer en `E` quand aucun élément x_i ne correspond aux arguments, ou en ajoutant des tests pour éviter ces appels inutiles en premier lieu (le coût des deux options est similaire). Il faut cependant prendre soin d'émettre un appel initial pour une taille une puissance de 2, qui n'est maintenant plus nécessairement le nombre de valeurs fournies.
Une alternative un peu différente et plus directe consiste à maintenir une pile (par exemple implémentée par une `list`) pour stocker l'ensemble des arbres t_i d'un certain niveau, puis à dépiler ceux-ci deux-par-deux afin de construire le niveau suivant, en traitant séparément le cas d'un éventuel arbre seul (qui est alors « virtuellement » fusionné avec `E`, c'est à dire directement transféré au niveau suivant).
On propose une implémentation de chaque approche :

```
let construire2 x =
    let s = Array.length x in
    let rec nextpow n = if n < 1 then 1 else 2 * (nextpow (n/2)) in
    let rec cons n i = if i >= s then E else
        if n = 1 then N (E, x.(i), E)
        else let tl = cons (n/2) (2*i) in
             let tr = cons (n/2) (2*i+1) in
             fusion (tl, tr)
    in cons (nextpow s) 0

let construire2' x =
    let n = Array.length x in
    let rec make_t0 acc i =
        if i = n then acc
        else make_t0 ((N (E, x.(i), E))::acc) (i + 1)
    in
    let rec ti2tip1 acc = function
        | [] -> acc
        | todd::[] -> todd::acc
        | ti::ti'::tl -> ti2tip1 (fusion (ti,ti')::acc) tl
    in
    let rec loop = function
        | [] -> assert false
        | [ti] -> ti
        | ti -> loop (ti2tip1 [] ti)
    in loop (make_t0 [] 0)
```

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