TD Option informatique *n*°2
Un algorithme de tri

Présentation

Motivation

Trier des données est un problème récurrent dans tous les systèmes d'information. Dans une application réelle, les données que l'on veut trier ne sont pas quelconques mais suivent une certaine distribution aléatoire, qui est loin d'être uniforme : ainsi, il est fréquent que les données soient déjà presque triées. Mais de nombreux algorithmes de tris, même les plus performants, atteignent leur complexité maximale lorsque les données sont déjà triées. C'est une situation très regrettable.

Le but de ce problème est d'étudier un algorithme de tri, proche du tri par tas mais présentant avec lui quelques différences significatives et notamment des performances intéressantes lorsque les données qu'il reçoit sont presque triées.

Dans tout le problème, on triera, par ordre croissant, des valeurs entières. La généralisation aux autres types ne pose pas de problème. Dans toutes les questions de complexité en temps, la mesure de complexité à considérer est le nombre de comparaisons par la relation d'ordre $\leq$ (ou par $\geq$, ou par $=$) entre entiers.

Notations et préliminaires

Dans la suite, si un objet mathématique est noté t, on notera `t` l'objet Caml qui l'implante et, si t appartient à un ensemble T implanté en Caml par le type `T`, on écrira de manière équivalente $t \in T$ ou `t : T`. Par exemple, pour signifier que l désigne une liste d'entiers, on notera `l : int list`.

Etant donné deux fonctions f et g à valeurs positives, on note :

- $f = O(g)$ pour exprimer qu'il existe une constante C telle que, pour tout n suffisamment grand, $f(n) \leq Cg(n)$;
- $f = \Omega(g)$ pour exprimer qu'il existe une constante $C > 0$ telle que pour, tout n suffisamment grand, $f(n) \geq Cg(n)$;
- $f = \Theta(g)$ pour exprimer qu'on a $f = O(g)$ et $f = \Omega(g)$.

On tiendra compte dans la suite que la structure à trier n'est pas un ensemble, car le même élément peut être répété plusieurs fois. Ainsi on sera amené à manipuler en Caml des listes ou des tableaux dans lesquels un même élément peut apparaitre plusieurs fois, et des arbres dans lesquels la même valeur peut étiqueter plusieurs sommets différents. Dans la suite, lorsqu'il s'agira de déterminer le minimum d'une telle structure, il pourra être atteint plusieurs fois ; de même, lorsqu'on triera ou "réunira" deux telles structures, ce sera toujours en tenant compte des répétitions, c'est-à-dire sans perte d'éléments. Ainsi, par exemple, pour les listes $l_1 = (7, 4, 2, 8, 2, 7, 3)$ et $l_2 = (5, 2, 3, 9)$, le minimum de l_1 est 2, trier l_1 consiste à renvoyer la liste `[2;2;3;4;7;7;8]` et "réunir" l_1 et l_2 consiste à renvoyer la liste `[7;4;2;8;2;7;3;5;2;3;9]`.

De manière générale, lorsqu'on dira que deux structures de données contiennent les mêmes éléments, ce sera toujours en tenant compte des répétitions. Par exemple les listes `[1;2;2]` et `[2;1;2]` contiennent les mêmes éléments mais pas les listes `[3;4;4]` et `[4;3;3]`.

I. Algorithme sur des arbres

I.A Tas binaires

On appelle *arbre* un arbre binaire étiqueté par des éléments de $\mathbb{N}$. Un tel arbre est implanté en Caml à l'aide de la déclaration de type suivante :

```
type arbre =
  | Vide
  | Noeud of int * arbre *arbre ;;
```

On définit la *hauteur* et la *taille* (appelée aussi nombre d'éléments) d'un arbre a, notées respectivement $\mathrm{h}(a)$ et $|a|$ par induction sur la structure de l'arbre :

- haut(`Vide`)=0 et haut(`Noeud(x,a1,a2)`)=1+max{haut(`a1`),haut(`a2`)} (Vide est de hauteur 0, faisons avec) ;
- $|\texttt{Vide}| = 0$ et $|\texttt{Noeud(x, a1, a2)}| = 1 + |a1| + |a2|$.

pour tous arbres binaires `a1` et `a2` et tout entier `x`.

L'entier `x` et les arbres `a1` et `a2` sont appelés respectivement la *racine*, le *fils gauche* et le *fils droit* de l'arbre `a`.

On dit que deux arbres *ont mêmes éléments* s'ils ont les mêmes ensembles d'étiquettes et que chaque étiquette présente apparait le même nombre de fois dans chacun des arbres.

On dit qu'un arbre binaire est *parfait* s'il s'agit de l'arbre vide `Vide`, ou s'il est de la forme `Noeud(x,a1,a2)` où `a1` et `a2` sont deux arbres parfaits de même hauteur.

On dit qu'un arbre binaire est un *tas binaire parfait* (ou simplement un *tas parfait*) si c'est un arbre parfait et que la valeur étiquetant chaque nœud de l'arbre est inférieure ou égale à celle de ses fils.

On dit qu'un arbre est un *quasi-tas* si c'est un arbre de la forme `Noeud(x,a1,a2)` et que `a1` et `a2` sont des tas binaires parfaits de même taille : aucune contrainte d'ordre n'est donc imposée sur l'étiquette de la racine `x`.

Etant donné un arbre non vide a, on note $\min_{\mathcal{A}}(a)$ le minimum des éléments qu'il contient.

I.A.1 Pour $k \in \mathbb{N}$, on note m_k la taille d'un arbre binaire parfait de hauteur k. Déterminer m_k pour tout $k \in \mathbb{N}$. On justifiera la réponse en exprimant m_{k+1} en fonction de m_k.

I.A.2 Ecrire la fonction `min_tas : arbre` $\rightarrow$ `int` telle que pour tout tas binaire parfait `a` non vide, `(min_tas a)` renvoie $\min_{\mathcal{A}}$(`a`). On fera en sorte que la complexité soit constante.

I.A.3 Ecrire la fonction `min_quasi : arbre` $\rightarrow$ `int` telle que pour tout quasi-tas `a`, `(min_quasi a)` renvoie $\min_{\mathcal{A}}$(`a`) en temps constant.

I.A.4 Ecrire la fonction `percole : arbre` $\rightarrow$ `arbre` telle que `(percole a)` renvoie `a` si `a` est l'arbre vide et, si `a` est un quasi-tas, renvoie un tas binaire parfait contenant les mêmes éléments. Donner la complexité dans le cas le pire en fonction de la hauteur k du quasi-tas `a`.

I.B. Décomposition parfaite d'un entier

L'algorithme de tri que l'on va étudier repose sur une propriété remarquable des nombres m_k obtenus en question A.1. Étant donné un entier naturel r, on dit qu'un r-uplet $(k_1, \ldots, k_r)$ d'entiers naturels non-nuls *vérifie la propriété QSC* (pour "quasi strictement croissant") lorsqu'on a $k_1 \leq k_2 < k_3 < \cdots < k_r$.

Remarques :

- Pour $r = 0$, on obtient qu'il existe un unique 0-uplet vérifiant la propriété QSC : l'unique 0-uplet, noté ().
- Pour $r = 1$, on obtient que tous les 1-uplets (k_1) vérifient la propriété QSC.
- Pour $r = 2$, on obtient qu'un 2-uplet (un couple) (k_1, k_2) vérifie la propriété QSC si et seulement si $k_1 \leq k_2$.

La propriété remarquable des nombres m_k qui nous intéressera est alors la suivante : *pour tout entier naturel non nul n, il existe un unique entier r et un unique r-uplet $(k_1, \dots, k_r)$ d'entiers naturels non nuls vérifiant la propriété QSC et tel que $n = m_{k_1} + \cdots + m_{k_r}$ (cette somme étant nulle pour $r = 0$).*

Une telle écriture $n = m_{k_1} + \cdots + m_{k_r}$, où $(k_1, \dots, k_r)$ vérifie la propriété QSC est appelée une *décomposition parfaite* de n. Par exemple, les entiers de 1 à 5 admettent les décompositions parfaites suivantes : $1 = m_1$, $2 = m_1 + m_1$, $3 = m_2$, $4 = m_1 + m_2$, $5 = m_1 + m_1 + m_2$.

On peut remarquer que, du fait de la stricte croissance de la suite d'entiers $(m_k)_{k\in\mathbb{N}}$, un r-uplet d'entiers naturels $(k_1, \dots, k_r)$ vérifie la propriété QSC si et seulement si le r-uplet $(m_{k_1}, \dots, m_{k_r})$ vérifie également cette propriété.

L'unicité d'une décomposition parfaite ne nous préoccupe pas ici (on l'admettra donc), mais seulement son existence. Plus précisément, l'outil dont nous aurons besoin par la suite est un algorithme récursif d'obtention d'une décomposition parfaite.

I.B.1 Donner la décomposition parfaite des entiers $6, 7, 8, 9, 10, 27, 28, 29, 30, 31, 100$ et 101.

I.B.2 Soit n un entier naturel admettant une décomposition parfaite de la forme $n = m_{k_1} + \cdots + m_{k_r}$.
Montrer qu'alors $n + 1$ admet une décomposition parfaite de la forme

$$n + 1 = \begin{cases} m_{k_1+1} + (m_{k_3} + \cdots + m_{k_r}) & \text{si } r \geq 2 \text{ et } k_1 = k_2 \\ m_1 + m_{k_1} + \cdots + m_{k_r} & \text{sinon.} \end{cases}$$

I.B.3 Ecrire la fonction `decomp_parf : int -> int list` telle que, pour out entier naturel n, `(decomp_parf n)` renvoie la liste $(m_{k_1}, \dots, m_{k_r})$ des entiers apparaissant dans la décomposition parfaite de `n` (dans cet ordre). Cette fonction devra avoir une complexité temporelle $O(n)$.

I.C. Création d'une liste de tas

On appelle *liste de tas* une liste de couples de la forme (a, t) où a désigne un tas binaire parfait et $t = |a|$ est la taille de l'arbre a : il existe donc un entier naturel k tel que $t = m_k$.

Une liste de tas est implémentée en Caml par le type `(arbre * int) list`.

Etant donnée une liste de tas h de la forme précédente, on définit

- la *longueur* de h, notée $\mathrm{long}(h)$, par

$$\mathrm{long}(h) = \begin{cases} 0 & \text{si } h \text{ est la liste vide} \\ r & \text{si } h = ((a_1, t_1), \dots, (a_r, t_r)) \end{cases}$$

- la *taille* de h, notée $|h|$, par

$$|h| = \begin{cases} 0 & \text{si } h \text{ est la liste vide} \\ \underbrace{|a_1|}_{=t_1} + \cdots + \underbrace{|a_r|}_{=t_r} & \text{si } h = ((a_1, t_1), \dots, (a_r, t_r)) \end{cases}$$

- la *hauteur* de h, notée $\text{haut}(h)$, par

$$\text{haut}(h) = \begin{cases} 0 & \text{si } h \text{ est la liste vide} \\ \max(\text{haut}(h_1), \dots, \text{haut}(h_r)) & \text{si } h = ((a_1, t_1), \dots, (a_r, t_r)) \end{cases}$$

- le *minimum* de h, noté $\min_{\mathcal{H}}(h)$, par

$$\min{}_{\mathcal{H}}(h) = \begin{cases} +\infty & \text{si } h \text{ est la liste vide} \\ \min(\min_{\mathcal{A}}(a_1), \dots, \min_{\mathcal{A}}(a_r)) & \text{si } h = ((a_1, t_1), \dots, (a_r, t_r)) \end{cases}$$

Comme pour les arbres binaires, on dit que deux listes de tas *ont mêmes éléments* si les deux listes des arbres les constituant font apparaître exactement les mêmes étiquettes avec exactement le même nombre d'apparitions de chaque étiquette. De même, une liste l d'entiers naturels et une liste de tas constituée d'arbres dont les étiquettes appartiennent à $\mathbb{N}$ *ont mêmes éléments* si les deux structures font apparaître exactement les mêmes éléments avec le même nombre d'apparitions de chaque élément.

On dit qu'une liste de tas $h = ((a_1, t_1), \dots, (a_r, t_r))$ *vérifie la condition TC* (pour "tas croissants") si le r-uplet d'entiers naturels $(t_1, \dots, t_r)$ vérifie la propriété QSC. On peut remarquer qu'une liste de tas vérifie la condition TC si et seulement si $|h| = t_1 + \dots + t_r$ est une décomposition parfaite de $|h|$. En particulier, la liste de tas vide vérifie la condition TC ; on constate enfin que toute liste de tas de longueur 1, c'est-à-dire de la forme $h = ((a, |a|))$, vérifie la condition TC.

I.C.1 *Des dominations.*

I.C.1.a Si h est une liste non vide de tas non tous vides, a-t-on nécessairement $\text{haut}(h) = O(\log_2(|h|))$? A-t-on nécessairement $\text{long}(h) = O(\log_2(|h|))$? Justifier.

I.C.1.b Mêmes questions si h est une liste de tas vérifiant la condition TC.

I.C.2 Considérons un arbre réduit à sa racine (c'est à dire un couple $(a, 1)$ correspondant à un tas binaire parfait) et une liste de tas $h = ((a_1, t_1), \dots, (a_r, t_r))$ vérifiant la condition TC. Si l'on ajoute le couple $(a, 1)$ en tête de la liste h, on obtient bien une liste de tas $(a, 1) :: h = ((a, 1), (a_1, t_1), \dots, (a_r, t_r))$, mais qui ne vérifie peut-être plus la condition TC. L'objectif de cette question consiste à concevoir, en utilisant les outils mis en oeuvre dans les question précédentes, un algorithme qui construit une liste de tas h' ayant les mêmes éléments que $(a, 1) :: h$ telle que h' vérifie la condition TC.

I.C.2.a On considère $h_1 = ((a_1^1, 1), (a_1^2, 3), (a_1^3, 7))$ et $h_2 = ((a_2^1, 3), (a_2^2, 3), (a_2^3, 7))$ deux listes de tas vérifiant la condition TC, où les arbres a_i^j sont donnés dans la figure 1.

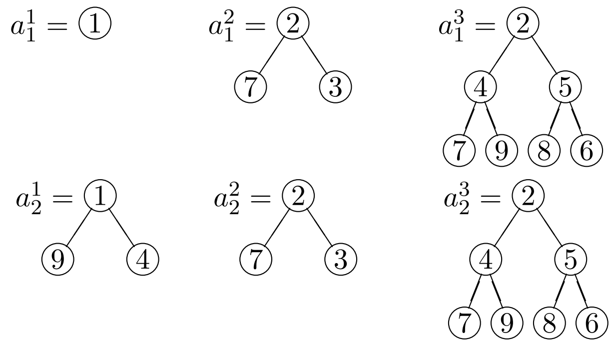


Figure 1

Expliquer de manière détaillée (à l'aide de représentation graphiques) comment on construit les listes de tas h_1' et h_2' lors de l'ajout de l'arbre a réduit à sa racine d'étiquette 8 dans chacune des listes de tas h_1 et h_2.

I.C.2.b Décrire le plus précisément possible un algorithme qui consiste à construire h' à partir d'un arbre a réduit à sa racine et d'une liste de tas h vérifiant la condition TC. On fera en sorte que cet algorithme ait une complexité dans le cas le pire $O(\text{haut}(a_1))$ (où a_1 est le premier tas de la liste h) et en $O(1)$ dans le cas le meilleur.

I.C.2.c Écrire la fonction `ajoute : int → (arbre * int) list → (arbre * int) list` telle que `(ajoute x h)` renvoie la liste de tas vérifiant la condition TC construite par l'algorithme de la question précédente à partir d'un arbre a réduit à sa racine `x` et une liste de tas h vérifiant la condition TC.

I.C.3 On définit la fonction suivante, de type `int list → (arbre * int) list` :

```
let rec constr_liste_tas l = match l with
  | [] -> []
  | x::r -> ajoute x (constr_liste_tas r) ;;
```

I.C.3.a Montrer que le coût de l'appel `(constr_liste_tas l)` pour une liste `l : int list` déjà triée de longueur n dans le cas le pire est en $O(n)$.

I.C.3.b Montrer que, pour une liste `l : int list` de longueur n, `(constr_liste_tas l)` a une complexité temporelle en $O(n \log_2(n))$ dans le cas le pire.[1]

I.D. Tri des racines

On dit qu'une liste de tas $h = ((a_1, t_1), \dots, (a_r, t_r))$ *vérifie la condition RO* (pour "racines ordonnées") si les tas présents dans la liste apparaissent par ordre croissant de leurs racines ou, ce qui est équivalent, par ordre croissant de leurs minimums : $\min_{\mathcal{A}}(a_1) \leq \dots \leq \min_{\mathcal{A}}(a_r)$.

On considère une liste de tas $h = ((a_1, t_1), \dots (a_r, t_r))$ vérifiant la condition TC et ne vérifiant pas nécessairement la condition RO. On veut réarranger les éléments apparaissant dans h de façon à obtenir une liste de tas h' vérifiant à la fois les conditions TC et RO. Si l'on trie brutalement par insertion les tas de h dans l'ordre croissant des racines, on risque de perdre la condition TC. On va donc mettre en place un tri s'inspirant du tri par insertion mais consistant à échanger les racines des tas présents dans h plutôt que les tas eux-mêmes.

I.D.1 Ecrire une fonction `echange_racines : arbre → arbre → (arbre * arbre)` de complexité constante telle que si `a1` et `a2` sont deux arbres binaires non vides, `(echange a1 a2)` renvoie le couple d'arbres passés en argument en se contentant d'échanger les étiquettes de leurs racines.

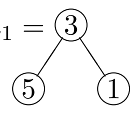
I.D.2 On considère une liste de tas non vide $h = ((a_1, t_1), \dots, (a_r, t_r))$ vérifiant la condition RO et un quasi-tas a de taille t. Montrer que :

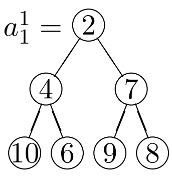
(a) si $\min_{\mathcal{A}}(a) \leq \min_{\mathcal{A}}(a_1)$, alors `(percole a, t)::h` est une liste de tas vérifiant RO ;

(b) si $\min_{\mathcal{A}}(a) > \min_{\mathcal{A}}(a_1)$ et si on pose `(b,b1)=(echange_racines a a1)`, alors `b` est un tas binaire parfait, `b1` est un quasi-tas et $\min_{\mathcal{A}}(\texttt{b}) = \min_{\mathcal{A}}(a_1) \leq \min_{\mathcal{A}}(\texttt{b1})$.

I.D.3 On examine maintenant trois exemples de couples (a, h) pour lesquels a est un quasi-tas et h est une liste de tas non vide vérifiant la condition RO. On souhaite à chaque fois faire évoluer la liste de tas $(a, |a|) :: h$ jusqu'à obtenir une liste de tas vérifiant la condition RO, *en ne s'autorisant pour seules opérations que d'éventuelles permutations entre des étiquettes d'un même arbre ou entre des étiquettes de deux arbres distincts* (aucune modification de la forme ou de la taille d'aucun arbre en jeu n'est autorisée).

Les couples considérés sont notés (a_1, h_1), (a_2, h_2) et (a_3, h_3) avec $h_1 = ((a_1^1, 7))$, $h_2 = ((a_2^1, 7))$ et $h_3 = ((a_3^1, 3), (a_3^2, 7))$, où les arbres a_1, a_1^1, a_2, a_2^1 et a_3, a_3^1, a_3^2 sont donnés figure 2.

$a_1 =$ 

$a_1^1 =$ 

1. On peut en fait montrer que cette complexité est un $\Theta(n)$ mais cela n'est pas demandé.

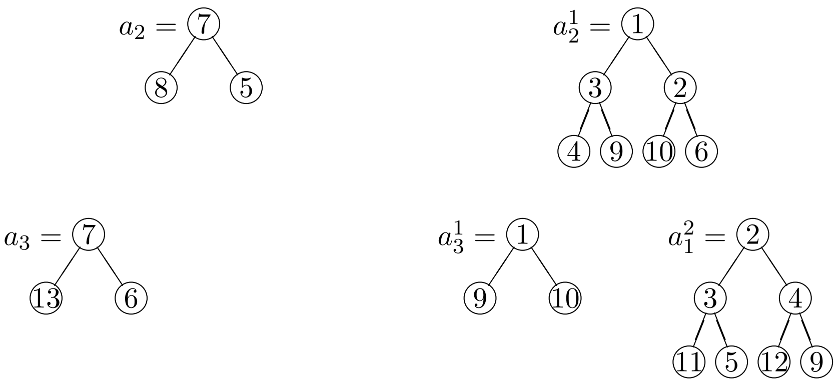


Figure 2

Pour chacun de ces trois couples, détailler (à l'aide de représentations graphiques) les étapes de la transformation de la liste $(a, |a|) :: h$ en une liste de tas vérifiant la condition RO. Chaque étape devra être clairement identifiée comme faisant appel à un procédé précédemment décrit.

I.D.4 Décrire et justifier le plus précisément possible un algorithme qui, à partir d'un quasi-tas a, de sa taille t et d'une liste de tas h, renvoie une liste de tas h' identique à $(a, t) :: h$ à permutation près des étiquettes des arbres et tel que si h vérifie la condition RO alors h' vérifie la condition RO.

Montrer que sa complexité en temps est $O(1)$ si a est un tas non vide, que la liste de tas h vérifie la condition et que $\min_{\mathcal{A}}(a) \leq \min_{\mathcal{H}}(h)$.

Montrer que sa complexité en temps est $O(k + r)$ où $k = \max\{\text{haut}(a), \text{haut}(h)\}$ et $r = \text{long}(h)$.

I.D.5 Ecrire la fonction
`insere_quasi : arbre → int → (arbre * int) list → (arbre * int) list`
telle que `(insere_quasi a t h)` renvoie la liste de tas vérifiant la condition RO construite par l'algorithme de la question précédente à partir d'un quasi-tas a de taille t et une liste de tas h vérifiant la condition RO.

I.D.6 Ecrire la fonction `tri_racines : (arbre * int) list → (arbre * int) list` transformant une liste de tas h supposée vérifier la condition TC en une liste h' de vérifiant à la fois la condition RO et la condition TC et telle que h et h' aient les mêmes éléments. Cette fonction devra avoir une complexité temporelle en $O((\log_2 |h|)^2)$.

I.E. Extraction des éléments d'une liste de tas

On souhaite dans cette sous-partie récupérer une liste d'étiquettes à partir d'une liste de tas vérifiant les propriétés précédentes. Soit $h = (\texttt{Noeud}(x, a_1, a_2), t) :: h'$ une liste de tas non vide vérifiant RO et TC. Pour supprimer x de h, si a_1 et a_2 ne sont pas vides, il suffit de construire $h'' = (\texttt{insere\_quasi}\ a_1\ |a_1|\ (\texttt{insere\_quasi}\ a_2\ |a_2|\ h'))$, où $|a_1|$ et $|a_2|$ peuvent se calculer en temps constant.

I.E.1 Montrer que h'' vérifie RO et TC.

I.E.2 Donner la complexité temporelle de l'évaluation de $(\texttt{insere\_quasi}\ a_1\ |a_1|\ (\texttt{insere\_quasi}\ a_2\ |a_2|\ h'))$ dans le cas le pire, sous la forme $O(f(|h|))$ pour une fonction f que l'on précisera.

I.E.3 Ecrire la fonction `extraire : (arbre * int) list → int list` prenant en argument une liste de tas h vérifiant les conditions RO et TC et renvoyant la liste triée des éléments de h en utilisant les idées ci-dessus. Cette fonction devra avoir une complexité temporelle en $O(|h| \log_2 |h|)$ dans le pire des cas.

I.F. Synthèse

I.F.1 Ecrire la fonction `tri_lisse: int list` $\to$ `int list` qui trie une liste en construisant une liste de tas intermédiaire vérifiant RO et TC avant d'en extraire les éléments.

I.F.2 Montrer que la complexité de cette fonction est en $O(n \log_2 n)$ où n est la longueur de la liste donnée en argument.

I.F.3 Déterminer la complexité temporelle de la fonction `tri_lisse` dans le cas particulier où la liste passée en argument est déjà triée.

II.Implantation dans un tableau

On s'intéressera dans cette partie à la complexité spatiale de certaines fonctions. Par complexité spatiale d'une fonction, on entend ici la quantité de mémoire dont elle a besoin pour s'exécuter, *la place utilisée par les données qu'elle reçoit en argument n'étant pas prise en compte*. Dit autrement, c'est la quantité de mémoire minimum qui doit rester disponible sur l'ordinateur au moment de l'appel de la fonction pour que son exécution ne provoque pas une erreur de capacité mémoire.
Le but de cette partie est de proposer un algorithme avec la même complexité temporelle que `tri_lisse` et une meilleure complexité spatiale.

II.A. Justifier brièvement que la complexité spatiale de `tri_lisse` sur une liste l de longueur n est en $\Omega(n)$.

Au lieu d'utiliser comme précédemment une structure d'arbres persistante, on va utiliser une structure impérative : on représentera des arbres sous forme d'une partie d'un tableau t : si a est un arbre de taille k, on le représente dans les k cases de t commençant à l'indice p comme suit :

- si a est vide, la représentation de a ne nécessite aucune place ;
- si a est de la forme `Noeud`(x, a_1, a_2), où a_1 et a_2 sont de tailles respectives k_1 et k_2, on met x dans la case p du tableau, puis on représente a_1 dans les k_1 cases commençant à l'indice $p+1$, puis on représente a_2 dans le taleau t dans les k_2 cases commençant à l'indice $p+1+k_1$ (cf figure 3).

$\cdots$	x	représentation de a_1	représentation de a_2	$\cdots$

$\uparrow$ p $\quad$ $\uparrow$ $p+1$ $\quad$ $\uparrow$ $p+k_1$ $\quad$ $\uparrow$ $p+k_1+1$ $\quad$ $\uparrow$ $p+k_1+k_2$

Figure 3

Au lieu de trier une liste d'entiers, on va trier un tableau d'entiers. Ce tableau servira à la fois à représenter les donnés initiales et les tas que nous manipulerons. Le tableau sera alors trié par échanges successifs.
Par la suite, tous les arbres que l'on représentera seront ou bien l'arbre vide, ou bien des quasi-tas (qui pourront éventuellement être des tas). On définit le type enregistrement suivant pour représenter l'arbre vide ou un quasi-tas stocké dans un tableau :

```
type tasbin = {donnees : int array ; pos : int ; taille : int} ;;
```

Les champs `donnees`, `pos` et `taille` d'un tel enregistrement contiennent respectivement le tableau où sont stockés les éléments, la position de la racine dans le tableau et le nombre d'éléments du quasi-tas (si ce nombre d'éléments est nul, le tableau et la position n'ont aucune importance).
Si le tableau stocké dans le champ `donnees` de cet enregistrement est le tableau t à trier, dont la consommation mémoire n'est pas comptée, la place mémoire prise par un élément de type `tasbin` est constante.
Par la suite, tous les éléments `t : tasbin` que nous manipulerons partageront le même tableau t.`donnees` sous-jacent, qui sera le tableau à trier.

II.B. Ecrire les fonctions `fg : tasbin` → `tasbin` et `fd : tasbin` → `tasbin` telles que si `a` représente un quasi-tas, `(fg a)` et `(fd a)` retournent respectivement une représentation de son fils-gauche et son fils-droit. Ces fonctions devront avoir une complexité constante.

II.C. Ecrire les fonctions `min_tas_array` et `min_quasi_array`, de type `tasbin` → `int` qui prennent en argument respectivement une représentation d'un tas binaire parfait non vide et une représentation d'un quasi-tas binaire et qui renvoient le minimum de leurs éléments.

II.D. Ecrire la fonction `percole_array : tasbin` → `unit` tel que si `a` est un quasi-tas, `(percole_array a)` échange des éléments dans le tableau `a.donnees` de façon que `a` devienne un tas avec préservation de l'ensemble des étiquettes aux répétitions près. Si `a` est un tas vide, `(percole_array a)` ne fait rien.

Comme précédemment, on s'intéressera à des listes de tas. De plus, tous les éléments de type `tasbin list` que nous manipulerons seront soit la liste vide soit des listes de la forme $(a_1, \ldots, a_r)$ telles que pour tout $i \in \{1, \ldots, r-1\}$, $a_i.\texttt{pos} + a_i.\texttt{taille} = a_{i+1}.\texttt{pos}$ et que $a_{r-1}.\texttt{pos} + a_{r-1}.\texttt{taille} = n$ où n est la longueur du tableau sous-jacent. La place mémoire occupée par une liste de tas est alors linéaire en sa longueur (et non en la longueur du tableau contenant ses éléments). Dans le cas où tous les éléments du tableau sont représentés dans la liste de tas, on aura de plus $a_1.\texttt{pos} = 0$.

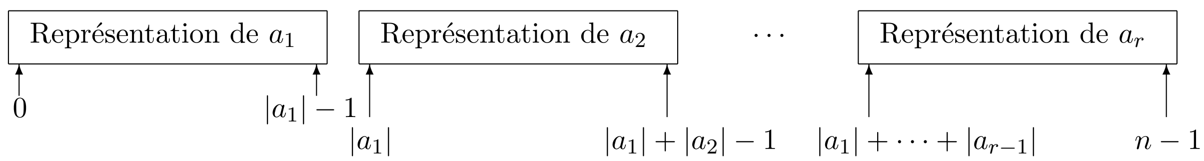


Figure 4

II.E. Etant donné un tableau `t`, on va construire un `h : tasbin list` en parcourant `t` de droite à gauche de la façon suivante : on démarre avec `h` vide et, pour k allant de $n-1$ à 0 inclus, on ajoute l'élément situé à l'indice k à `h`, de façon à garantir que la condition TC est préservée. A la fin, tous les éléments de `t` sont représentés dans `h`. On trie alors `h`, puis on extrait successivement le minimum de `h`.
Ecrire la fonction `ajoute_array : int array` → `int` → `tasbin list` → `tasbin list` analogue à la fonction `ajoute` précédemment définie et telle que `(ajoute_array d p h)` ajoute l'élément d'indice `p` du tableau `d` à `h`. On supposera que `h` est vide ou que la position du premier tas de `h` est `p+1`.

On définit alors la fonction `constr_liste_tas_array : int array` → `tasbin list`, analogue à la fonction `constr_liste_tas`, comme suit :

```
(* constr_liste_tas_aux : int array -> int -> tasbin list
           renvoie une liste de tas contenant les \'el\'ements d'indice >= p de d
           *)
let rec constr_liste_tas_aux d p =
     if p = Array.length d then []
     else
       ajoute_array d p (constr_liste_tas_aux d (p+1))
     ;;
let constr_liste_tas_array d = constr_liste_tas_aux d 0;;
```

II.F. Ecrire la fonction `echange_racines_array : tasbin` → `tasbin` → `unit` échangeant les racines des tas qui lui sont donnés en argument.

II.G. Ecrire la fonction `insere_quasi_array : tasbin` → `tasbin list` → `tasbin list` analogue de la fonction `insere_quasi`.

II.H. Ecrire la fonction `tri_racines_array : tasbin` → `tasbin list` → `tasbin list` analogue de la fonction `tri_racines`.

II.I. Ecrire la fonction `extraire_array : tasbin list` → `unit` analogue de la fonction `extraire`.

II.J. Ecrire la fonction `tri_lisse_array : int array` → `unit` triant un tableau en utilisant les fonctions précédentes.

II.K. Quelle est la complexité temporelle de `tri_lisse_array` dans le cas le pire ?

II.L. Quelle est la complexité temporelle de `tri_lisse_array` pour un tableau déjà trié ?

II.M. Quelle est la complexité spatiale de `tri_lisse_array` dans le cas le pire ?