INF431

Arbres Binaires Équilibrés
CORRIGÉ
Version: 1989:2422M

N.B. Dans cette P.C, quand il est question d'écrire une fonction, celle-ci est demandée impérativement en pseudo-code. Les sous-sections 1.1 et 1.2 sont totalement indépendantes et peuvent être traitées dans n'importe quel ordre.

1 Arbres de Braun

Un arbre binaire de racine a, de sous-arbre gauche T_1 et de sous-arbre droit T_2 est noté $< T_1, a, T_2 >$. L'arbre vide est noté par $<>$ et pour un arbre réduit à un seul sommet a, on utilisera la notation simplifiée $< a >$. Les arbres de Braun sont des arbres binaires donnés par la définition récursive suivante. Un arbre binaire $< T_1, a, T_2 >$ est dit « de Braun » s'il est soit l'arbre vide, soit de la forme $< T_1, a, T_2 >$ où T_1, a et T_2 vérifient les propriétés suivantes :

1. T_1 et T_2 sont des arbres de Braun,
2. $|T_2| \leq |T_1| \leq |T_2| + 1$.

Comme cela a été vu en cours, ces arbres sont utilisés par exemple pour implémenter des files de priorité.

Question 1 Écrire une fonction prenant en entrée un arbre binaire et renvoyant `vrai` s'il est de Braun et `faux` sinon. Quelle est la complexité de votre fonction ? ◇

Solution. Une manière de procéder est de commencer par écrire une fonction qui retourne la taille de l'arbre si celui-ci est de Braun et -1 sinon.

fonction TAILLEEQUILIBREE(T)
si ESTVIDE(T)**renvoyer** 0
sinon
- $s \leftarrow$ TAILLEEQUILIBREE($T.fg$); $t \leftarrow$ TAILLEEQUILIBREE($T.fd$)
- **si** $s < 0$ **ou** $t < 0$ **ou** $t > s$ **ou** $s > t + 1$
- **alors renvoyer** -1
- **sinon renvoyer** $1 + s + t$

qui est appelée dans la fonction

fonction ESTDEBRAUN(T)
$s \leftarrow$ TAILLEEQUILIBREE(T)
si $s < 0$
alors renvoyer FAUX
sinon renvoyer VRAI

Cet algorithme a évidemment une complexité du même ordre que la taille de l'arbre auquel il s'applique. □

1.1 Tableaux de taille variable

La structure de données abstraite appelée "tableau de taille variable" ou "liste à accès aléatoire" est une structure hybride entre un tableau et une liste. Les manipulations qu'elle doit permettre sont d'une part les manipulations standard sur les tableaux comme accéder à un élément quelconque ou le modifier

et d'autre part les manipulations standard des listes comme ajouter/enlever un élément en début ou fin de liste. On utilise typiquement ce type de structure quand on veut disposer d'un tableau dont la taille varie de manière dynamique. Nous allons voir ici une implémentation de cette structure au moyen d'arbres de Braun.

Question 2 Montrer qu'il existe un unique arbre de Braun à n sommets. Cet arbre est noté B_n. Montrer que B_n s'obtient à partir de B_{n-1} en rajoutant un sommet. ◇

Solution. Cela se montre par une récurrence immédiate sur n (avec l'hypothèse de récurrence : "il existe un unique arbre de Braun de taille i pour $0 \leq i \leq n$") et provient du fait que le fils gauche d'un tel arbre est nécessairement un arbre de Braun à $\lceil \frac{n-1}{2} \rceil$ sommets alors que celui de droite est un arbre de Braun à $\lfloor \frac{n-1}{2} \rfloor$ sommets.

La deuxième propriété se montre elle aussi immédiatement par récurrence sur n en remarquant puis en utilisant le fait que $B_{2n+1} =< B_n, r, B_n >$ et $B_{2n+2} =< B_{n+1}, r, B_n >$ où r désigne la racine. □

De la question précédente, on déduit une numérotation récursive des sommets des arbres de Braun : on numérote par 0 l'unique sommet de B_1 et les numéros des sommets de B_n correspondent aux numéros des sommets de B_{n-1} sauf pour le seul sommet qui n'appartient pas à B_{n-1}, et celui-ci est numéroté $n-1$.

Question 3 Donner la numérotation des sommets de l'arbre de Braun de taille 15. ◇

Solution. En commençant par B_1, puis en rajoutant un sommet pour obtenir B_2 et ainsi de suite, on obtient la figure 1.

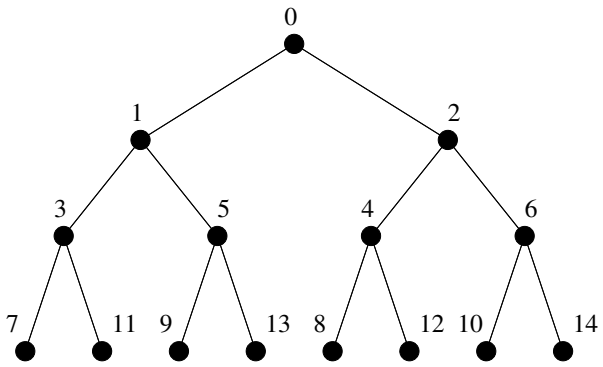


FIGURE 1 – Arbre de Braun de taille 15 avec sa numérotation.

On va maintenant utiliser cette numérotation des sommets des arbres de Braun pour implémenter des tableaux de taille variable sous la forme d'un arbre de Braun, un tableau $\texttt{tab}[0]$,$\texttt{tab}[1], \ldots, \texttt{tab}[n-1]$, étant représenté par l'arbre de Braun B_n contenant dans son sommet numéroté i la valeur $\texttt{tab}[i]$. Une telle implémentation d'un tableau de taille n est réalisée au moyen du couple (B, n) où B est l'arbre de Braun représentant le tableau et n est la taille de l'arbre. On appellera dans ce qui suit le couple (B, n) un *tableau de Braun*.

Question 4 Ecrire une fonction ELEMENT(B, i) qui renvoie l'élement de numéro i dans B. ◇

Solution. La numérotation a une description récursive qui peut se formaliser de la manière suivante. Pour comprendre où se situe un sommet de numéro n dans un arbre de Braun $B_{n'}$ avec $n' > n$, il suffit de comprendre à quel endroit il a été rajouté dans l'arbre B_n. Notons $m = \lfloor \frac{n-1}{2} \rfloor$. Si n est pair, alors B_n est de la forme $< B_{m+1}, r, B_m >$ et le sommet numéroté n va être rajouté dans le sous-arbre droit à l'endroit où le sommet numéroté m serait rajouté à B_m. Si n est impair, alors B_n est de la forme $< B_m, r, B_m >$ et le sommet numéroté n va être rajouté dans le sous-arbre gauche à l'endroit où le sommet de numéro m serait rajouté à B_m. De cette description, on déduit immédiatement le pseudo-code suivant

fonction ELEMENT(B, i)
si $i = 0$ **alors renvoyer** $(B.racine)$
sinon $\begin{cases} j \leftarrow \lfloor \frac{i-1}{2} \rfloor \\ \textbf{si } \text{ESTPAIR}(i) \textbf{ alors renvoyer } \text{ELEMENT}(B.fd, j) \\ \textbf{sinon renvoyer } \text{ELEMENT}(B.fg, j) \end{cases}$ □

Question 5 Décrire brièvement deux algorithmes pour rajouter ou supprimer un élément en queue d'un tableau de Braun. Quelle est leur complexité ? Voyez-vous à quoi sert le fait de stocker n avec l'arbre de Braun ? ◇

Solution. Pour supprimer l'élément en queue de tableau de Braun B_n, on utilise la description récursive de la numérotation qui précède pour retrouver cet élément : on examine les sous-arbres gauche et droit de B_n, on va ensuite dans le sous-arbre droit si les deux sous-arbres sont de la même taille et à gauche sinon, et enfin, on itère le procédé jusqu'à trouver une feuille que l'on remplace par l'arbre vide. Pour savoir ou placer un nouvel élément, on fait l'inverse : on examine les sous-arbres gauche et droit B_n, on va dans le sous-arbre gauche si les deux sous-arbres sont de la même taille et à droite sinon, et on itère le procédé jusqu'à tomber sur un arbre vide que l'on remplace par l'élément en question. Ces opérations ont la même complexité que la hauteur de l'arbre, soit $O(\log n)$.

Question 6 Ecrire la fonction RAJOUTEENTETE(T, e) ajoutant un élément en début du tableau de Braun et renvoyant le tableau de Braun correspondant. Ecrire ensuite la fonction SUPPRIMETETE(T) supprimant le premier élément du tableau de Braun et renvoyant le tableau de Braun correspondant. ◇

Solution. Comme l'implémentation des files de priorité avec des arbres de Braun vue en cours, on place l'élément à la racine et l'ancienne racine est placée dans l'arbre droit, puis on échange les deux sous-arbres pour satisfaire la contrainte que le sous-arbre gauche a une taille supérieure ou égale à celle du sous-arbre droit. On obtient pour les mêmes raisons que dans le cours un arbre de Braun à la fin. Reste maintenant à vérifier que les éléments sont placés à la bonne place. Cela se fait en montrant dans un premier temps par récurrence sur n que la numérotation de l'arbre de Braun B_n représentant le tableau $\texttt{tab}[0]$,$\texttt{tab}[1], \ldots, \texttt{tab}[n-1]$, vérifie les conditions suivantes

(i) le sous-arbre gauche porte les éléments d'indice impair du tableau, et si l'on définit le tableau `gauche` comme $\texttt{gauche}[i] = \texttt{tab}[2i+1]$ alors l'arbre de Braun correspondant à ce sous-arbre gauche représente le tableau `gauche` ;

(ii) le sous-arbre droit porte les éléments d'indice pair du tableau à l'exception de l'indice 0, et si l'on définit le tableau `droit` comme $\texttt{droit}[i] = \texttt{tab}[2i+2]$ alors l'arbre de Braun correspondant à ce sous-arbre gauche représente le tableau `droit`.

La correction de l'algorithme se montre alors par récurrence sur la taille de l'arbre sur laquelle on l'applique. L'hypothèse de récurrence est ici « L'algorithme crée le bon tableau de Braun pour tous les tailles d'arbre allant de 0 à n ».

Cette hypothèse est évidemment vraie pour $n = 0$. Considérons maintenant le tableau de Braun de taille $n+1$ avec comme arbre de Braun associé B_{n+1}. Il représente un tableau $\texttt{tab}[0]$,$\texttt{tab}[1], \ldots,$ $\texttt{tab}[n]$, et on veut rajouter un élément e en début de tableau, de telle sorte que le nouveau tableau `tabN` représenté par cet arbre de Braun vérifie $\texttt{tabN}[0] = e$,$\texttt{tabN}[1] = \texttt{tab}[0], \ldots, \texttt{tabN}[n+1] = \texttt{tab}[n]$. L'ajout de $\texttt{tab}[0]$ au sous-arbre droit conduit alors en utilisant à la fois l'hypothèse de récurrence et la propriété (ii) plus haut à un arbre de Braun qui représente un tableau ayant pour éléments tous les éléments du tableau `tab` d'indice pair rangés de manière croissante. Ce sous-arbre est placé à gauche et il correspond donc aux éléments d'indice impair du tableau `tabN` rangés par indice croissant. On a donc bien comme souhaité : $\texttt{tabN}[1] = \texttt{tab}[0]$,$\texttt{tabN}[3] = \texttt{tab}[2], \ldots$.

Par ailleurs, le sous-arbre gauche de l'ancien arbre de Braun est placé comme sous-arbre droit du nouvel arbre de Braun. Il représentait le tableau ayant pour éléments tous les éléments du tableau d'indice pair rangés de manière croissante. Comme il est placé à droite il représente maintenant le tableau formé de tous les éléments du tableau `tabN` d'indice pair (à l'exception de $\texttt{tabN}[0]$) rangés par indice croissant. On a donc là aussi comme souhaité : $\texttt{tabN}[2] = \texttt{tab}[1]$,$\texttt{tabN}[4] = \texttt{tab}[3], \ldots$. Ceci montre

donc que l'algorithme crée le bon tableau de Braun quand la taille de l'arbre est $n+1$ et montre donc la correction de l'algorithme par récurrence.

On obtient ainsi le pseudo-code de la fonction suivante

fonction RAJOUTEENTETEREC(B, e)
si ESTVIDE(B) **alors renvoyer** $(< e >)$
sinon renvoyer $< \text{RAJOUTEENTETEREC}(B.fd, B.racine), e, B.fg) >$

fonction qui est appelée dans

fonction RAJOUTEENTETE$((B, n), e)$
$B \leftarrow \text{RAJOUTEENTETEREC}(B, e)$
renvoyer $(B, n+1)$

De la même manière on écrit la fonction SUPPRIMETETE dont la correction se montre de manière similaire (elle consiste essentiellement dans le cas simple de la suppression de l'élément de plus grande priorité d'une file de priorité implémentée avec un arbre de Braun vue en cours)

fonction FUSIONNE(T_1, T_2)
si ESTVIDE(T_2) **alors renvoyer** (T_1)
sinon renvoyer $< T_2, T_1.racine, \text{FUSIONNE}(T_1.fg, T_1.fd) >$

fonction qui est appelée dans

fonction SUPPRIMETETE$((B, n))$
si $n = 0$ **alors renvoyer** (B, n)
sinon si $n = 1$ **alors renvoyer** $(<>, 0)$
sinon $\begin{cases} B \leftarrow < B.fd, B.fg.racine, \text{FUSIONNE}(B.fg.fg, B.fg.fd) \\ \textbf{renvoyer } (B, n-1) \end{cases}$

□

1.2 Deux opérations qui s'effectuent de manière efficace sur un arbre de Braun

Calculer la taille d'un arbre binaire prend un temps linéaire en la taille de l'arbre. Dans le cas particulier d'un arbre de Braun, en raison des contraintes d'équilibrage, il est possible de faire bien mieux.

Question 7 Ecrire une fonction de complexité sous-linéaire pour calculer la taille d'un arbre de Braun. Quelle est sa complexité exacte ? ◇

Solution. L'idée est d'abord de remarquer que

$$\text{TAILLE}(B) = 1 + 2\text{TAILLE}(B.fd) + \text{RELIQUAT}(B.fg, \text{TAILLE}(B.fd))$$

où RELIQUAT(B, m) est une fonction qui n'est définie que pour des arbres de Braun B et $m \leq \text{TAILLE}(B) \leq m+1$ et vaut 0 si $m = \text{TAILLE}(B)$ et 1 sinon. Le point crucial est que cette dernière peut être calculée avec une complexité de l'ordre $O(\log |B|)$ quand B est un arbre de Braun.

Elle peut être obtenue de manière très simple en modifiant légèrement la fonction ELEMENT de la section précédente. Elle peut aussi être obtenue en faisant le raisonnement suivant qui est un bon exemple de comment utiliser les *préconditions* (mais il est à signaler que cela conduit essentiellement à la même écriture qu'en cherchant à modifier la fonction ELEMENT). Pour le raisonnement qui suit, nous allons faire l'hypothèse que la fonction RELIQUAT est toujours appelée sur des couples (B, m) appartenant à son domaine de définition, c'est à dire tels que (i) B est un arbre de Braun et (ii) $m \leq \text{TAILLE}(B) \leq m+1$. C'est ce que l'on appelle une *précondition*, c'est à dire une condition qui doit être vérifiée au moment de l'appel de la fonction, juste avant qu'elle ne soit exécutée. Nous vérifierons par la suite dans l'écriture des fonctions réalisant le calcul de la taille, que les appels de la fonction RELIQUAT vérifient toujours cette précondition. Posons $m' = \lfloor \frac{m-1}{2} \rfloor$. Ici, deux cas sont à considérer

1. m est pair et donc $m = 2m'+2$. Ici deux sous-cas sont possibles : soit le sous-arbre droit de B est de taille m' et dans ce cas le sous-arbre gauche est nécessairement de taille $m'+1$ et le reliquat vaut 0, soit le sous-arbre droit est de taille $m'+1$ et le sous-arbre gauche est donc nécessairement de même taille et le reliquat vaut 1. Notez bien que l'on a utilisé ici la précondition : B est un

arbre de Braun et $m \leq \text{TAILLE}(B) \leq m+1$. En effet, reprenons en détail le premier cas, c'est à dire quand la taille du sous-arbre droit est m'. Comme B est un arbre de Braun, deux cas sont possibles : soit le sous arbre gauche est de taille m', soit il est de taille $m'+1$. Le premier cas est impossible car B serait alors de taille $2m'+1$ mais sa taille serait alors inférieure à m, ce qui viole la deuxième partie de la précondition. On vérifie en revanche qu'une taille de $m'+1$ est possible pour le sous-arbre gauche. Le cas où le sous-arbre droit est de taille $m'+1$ est traité de manière similaire. Au final, on remarque que dans tous les cas, le reliquat vaut $\text{RELIQUAT}(B.fd, m')$.

2. m est impair et donc $m = 2m'+1$. Deux sous-cas sont possibles : soit le sous-arbre gauche est de taille m', dans ce cas le sous-arbre droit est de même taille et le reliquat vaut alors 0, soit le sous-arbre gauche est de taille $m'+1$ et le reliquat vaut 1. Cela se détecte en calculant $\text{RELIQUAT}(B.fg, m')$.

Toutes ces considérations conduisent à l'écriture récursive suivante

fonction $\text{TAILLE}(B)$
si $\text{ESTVIDE}(B)$ **alors renvoyer** 0
sinon
$m \leftarrow \text{TAILLE}(B.fd)$
renvoyer $1 + 2m + \text{RELIQUAT}(B.fg, m)$

avec la fonction RELIQUAT définie comme

fonction $\text{RELIQUAT}(B, m)$
si $m = 0$
alors
si $\text{ESTVIDE}(B)$ **alors renvoyer** 0
sinon renvoyer 1
sinon
$l \leftarrow \lfloor \frac{m-1}{2} \rfloor$
si $\text{ESTPAIR}(m)$ **alors renvoyer** $\text{RELIQUAT}(B.fd, l)$
sinon renvoyer $\text{RELIQUAT}(B.fg, l)$

Vérifions maintenant la correction de cette écriture récursive. On vérifie dans l'écriture récursive de TALLE que la précondition sur RELIQUAT est toujours vérifiée. On utilise une précondition supplémentaire portant cette fois-ci sur la fonction TAILLE : elle est appelée sur des instances de B qui sont des arbres de Braun. Bien entendu, on vérifiera également que les appels à TAILLE se font toujours dans des cas où la précondition est vérifiée. Dans l'écriture de TAILLE on fait appel à la fois à TAILLE que l'on appelle sur $B.fd$, et c'est bien un arbre de Braun puisque B était un arbre de Braun, et on appelle RELIQUAT. Dans l'appel de RELIQUAT sur le couple $(B.fg, m)$, on vérifie que les deux parties de la précondition sont elles-aussi bien vérifiées. En effet, la première l'est de manière immédiate parce que $B.fg$ est un arbre de Braun. La deuxième l'est parce que m est égal à la taille de $B.fd$ et que la condition sur m et la taille de $B.fg$ que l'on cherche à vérifier correspond à la condition d'équilibrage de l'arbre de Braun B.

Dans la fonction RELIQUAT on a deux appels à RELIQUAT dont il faut vérifier la précondition. La première partie de la précondition est immédiate, en effet B est un arbre de Braun et donc à la fois $B.fg$ et $B.fd$ le sont. La vérification de la seconde partie de la précondition revient essentiellement à rappeler la discussion plus haut.

Enfin, la fonction RELIQUAT est clairement de complexité $O(\log |B|)$ et on en déduit que la fonction TAILLE est de complexité $O(\log^2 |B|)$. □

On se pose maintenant la question de créer le plus efficacement possible un arbre de Braun contenant n copies d'un même élément e. Considérons la fonction récursive suivante qui réalise cette opération

fonction $\text{COPIE}(e, n)$
si $n = 0$ **alors renvoyer** $<>$
sinon
$m \leftarrow \lfloor \frac{n-1}{2} \rfloor$
si $\text{ESTPAIR}(n)$ **alors renvoyer** $< \text{COPIE}(e, m+1), e, \text{COPIE}(e, m) >$
sinon
$T \leftarrow \text{COPIE}(e, m)$
renvoyer $< T, e, T >$

Question 8 Quelle est la complexité de cette fonction ? ◇

Solution. Notons $f(n)$ la complexité de l'appel de la fonction COPIE(e, n). On a

$$\begin{aligned} f(2n+1) &\leq C + f(n) && (1)\\ f(2n+2) &\leq C + f(n) + f(n+1) && (2)\end{aligned}$$

pour une certaine constante C. Pour deviner quelle pourrait être l'ordre de grandeur de $f(n)$ on peut remarquer que dans (2) soit n, soit $n+1$ est impair. Supposons que ce soit n. On aurait alors en combinant (1) et (2) que

$$f(2n+2) \leq 2C + f\left(\frac{n-1}{2}\right) + f(n+1). \quad (3)$$

Dans le cas $n+1$ impair, on aurait

$$f(2n+2) \leq 2C + f(n) + f(n/2) \quad (4)$$

En d'autres termes on a approximativement une inégalité du type

$$g(n) \leq C' + g(n/2) + g(n/4)$$

pour laquelle on a envie de poser $h(t) = g(2^t) + C'$, auquel cas on aurait

$$h(t) \leq h(t-1) + h(t-2)$$

et l'on montrerait que $h(t)$ est majoré par $O(u_t)$ où u_t est la suite de Fibonacci. On pose $\phi = \frac{1+\sqrt{5}}{2}$ et on obtient $h(t) = O(\phi^t)$. En d'autres termes, cela suggère de montrer par récurrence que $f(n) \leq K\phi^{\log_2 n} = Kn^{\frac{\ln\phi}{\ln 2}}$. On pose $\alpha = \frac{\ln\phi}{\ln 2}$. Notons que $\frac{\ln\phi}{\ln 2} \approx 0.694$ et l'on aurait une complexité sous-linéaire. Nous allons aussi utiliser par la suite l'égalité

$$1 + 2^\alpha = 4^\alpha. \quad (5)$$

Il sera légèrement plus agréable de montrer par récurrence que g définie par $g(n) = 2C + f(n)$ vérifie

$$g(n) \leq Kn^\alpha \quad (6)$$

et d'en déduire que $f(n)$ vérifie la même inégalité. L'intérêt d'introduire un tel g est que cette fois ci on a les inégalités légèrement plus simples pour le cas n pair

$$g(2n+2) \leq g\left(\frac{n-1}{2}\right) + g(n+1). \quad (7)$$

et pour le cas n impair

$$g(2n+2) \leq g(n) + g(n/2) \quad (8)$$

Montrons maintenant (6) par récurrence pour un K bien choisi. L'hypothèse de récurrence est que $g(1), \ldots, g(2n-1), g(2n)$ vérifient tous $g(t) \leq Kt^\alpha$ pour $t \in \{1, 2, \ldots, 2n-1, 2n\}$. On va déjà choisir $K \geq \max(g(1), g(2)2^{-\alpha})$ pour que cette hypothèse de récurrence soit vraie pour $n = 1$. Supposons la donc vérifiée jusqu'à l'ordre n. Observons que si l'on impose la contrainte supplémentaire $K \geq C$ on peut écrire

$$\begin{aligned} g(2n+1) &\leq C + g(n)\\ &\leq C + Kn^\alpha\\ &\leq K(1 + n^\alpha)\\ &\leq K(2n+1)^\alpha \end{aligned}$$

où la dernière inégalité est une conséquence de $1 + x^\alpha \leq (1 + 2x)^\alpha$ qui est vraie pour $x \in [1, +\infty[$. Par ailleurs, en observant que soit (7), soit (8) sont vraies et en utilisant l'hypothèse de récurrence, on obtient

$$\begin{aligned} g(2n+2) &\leq \max\left(K\left(\frac{n-1}{2}\right)^\alpha + K(n+1)^\alpha, Kn^\alpha + Kn/2^\alpha\right) \\ &\leq K\left(\frac{n+1}{2}\right)^\alpha + K(n+1)^\alpha \\ &\leq K(2n+2)^\alpha \end{aligned}$$

en utilisant pour la dernière inégalité (5). Ceci montre que l'hypothèse de récurrence est encore vraie à l'ordre $n+1$ et termine la preuve.

Note importante : Plusieurs points sont à souligner ici. D'une part, il est hautement déconseillé d'écrire une fonction de création d'arbre de la sorte. Noter qu'il y a identité entre l'arbre gauche et droite quand on crée un arbre de Braun de taille impaire, cela provient de l'instruction effectuée dans la dernière ligne du pseudo-code

renvoyer $< T, e, T >$ □

En d'autres termes une modification ultérieure de le sous-arbre gauche impactera *aussi* le sous-arbre droit. Il est à noter ici que *toute* procédure de création d'arbre de complexité sous-linéaire aurait ce problème. En effet, la place mémoire occupé par un tel arbre serait alors nécéssairement sous-linéaire, par conséquent on aura nécéssairement deux sommets qui se partagent la même place en mémoire. Le problème mentionné précédemment provient de ce partage. Ce problème concerne bien évidemment l'amélioration proposé dans la question suivante.

Question 9 Proposez une amélioration de cette fonction qui soit de complexité $O(\log n)$. ◇

Solution. On peut remarquer que dans la solution précédente, le point qui rend la solution inefficace correspond à la récursion où n est pair auquel cas il faut à la fois créer l'arbre de Braun avec m éléments et celui avec $m+1$ éléments. Cela suggère fortement de créer ces deux arbres en *même temps*. Une fois que l'on a eu cette idée, l'écriture récursive suivante de la fonction COPIEDOUBLE(e, n) renvoyant le couple (COPIE$(e, n+1)$,COPIE(e, n)) s'en déduit immédiatement

fonction COPIEDOUBLE(e, n)
si $n = 0$ **alors renvoyer** $(< e >, <>)$
sinon
- $m \leftarrow \lfloor \frac{n-1}{2} \rfloor$
- $(T_1, T_2) \leftarrow$ COPIEDOUBLE(e, m)
- **si** ESTPAIR(n) **alors renvoyer** $(< T_1, e, T_1 >, < T_1, e, T_2 >)$
- **sinon renvoyer** $(< T_1, e, T_2 >, < T_2, e, T_2 >)$

On en déduit la fonction qui nous intéresse

fonction COPIE(e, n)
$(T_1, T_2) \leftarrow$ COPIEDOUBLE(e, n)
renvoyer T_2

Notons $f(n)$ la complexité de l'appel de la fonction COPIEDOUBLE(e, n). On a

$$f(n) \leq C + f\left(\frac{n-1}{2}\right)$$

On montre par une récurrence immédiate que la fonction $f(n)$ est de la forme $O(\log n)$. C'est aussi la complexité de l'appel de la fonction COPIE(e, n). □

2 Skew Heaps

On considérera maintenant des arbres binaires généraux qui portent en leur sommet des entiers et qui ont la propriété de *tas* : c'est à dire qu'en tout noeud, l'entier qui y est porté est inférieur ou égal à tous les éléments situés dans les sous-arbres gauche et droit. On va utiliser une notion d'équilibrage qui, bien qu'elle soit plus lâche que dans le cas des arbres de Braun, reste tout de même efficace. L'efficacité n'est pas mesurée ici par une complexité dans le pire cas (elle est mauvaise ici en fait) mais en prenant la moyenne sur toutes les opérations effectuées, c'est ce que l'on appelle la *complexité amortie*. Plus précisément, on montre avec cette notion que si l'on a un arbre de taille n, une succession de n insertions ou extractions de plus petit élément a une complexité totale de $O(n \log n)$ (et donc une complexité par opération ou complexité amortie de $O(\log n)$) même si certaines de ces opérations ont une complexité qui est plus grande que $O(\log n)$.

Plus généralement, la complexité amortie est adaptée aux structures de données dynamiques sur lesquelles sont répétées des opérations élémentaires (insertion, suppression, recherche par exemple) qui sont en général efficaces mais nécessitent de temps à autre de coûteuses réorganisations de la structure. On cherche alors à faire en sorte que les opérations coûteuses ne se produisent qu'après qu'un nombre suffisant d'opérations peu coûteuses aient été effectuées : l'effet d'amortissement ainsi obtenu peut permettre de garantir un bon coût amorti, c'est-à-dire un bon coût moyen par opération, quelle que soit la séquence valide d'opérations effectuées.

Décrivons maintenant en détail comment les opérations d'insertion et de suppression sont effectuées dans le tas. Celles-ci sont décrits par les équations récursives suivantes :

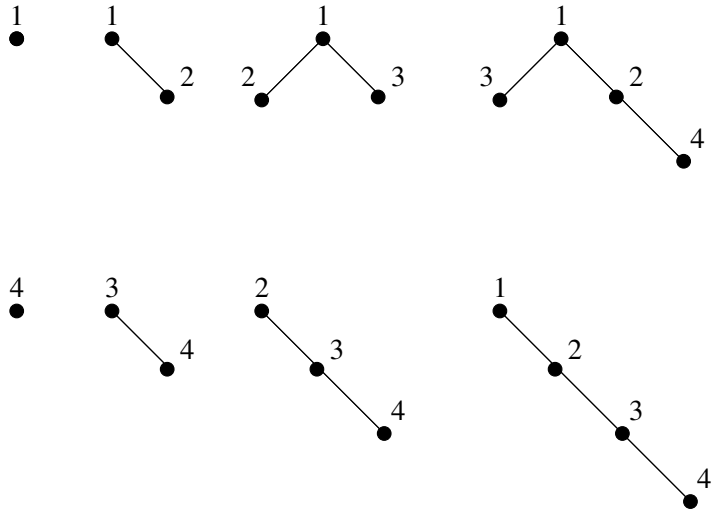
$$
\begin{aligned}
\textsc{extract}(< T_1, a, T_2 >) &= (a, \textsc{merge}(T_1, T_2)) \\
\textsc{insert}(a, T) &= \textsc{merge}(T, < a >)
\end{aligned}
$$

$$
\begin{aligned}
\textsc{merge}(T, <>) &= T \\
\textsc{merge}(<>, T) &= T \\
\textsc{merge}(T, U) &= \textsc{join}(T, U) \quad \text{si la racine de } T \text{ est plus petite que la racine de } U \\
\textsc{merge}(T, U) &= \textsc{join}(U, T) \quad \text{sinon}
\end{aligned}
$$

$$
\textsc{join}(< T_1, a, T_2 >, U) = < T_2, a, \textsc{merge}(T_1, U) >
$$

Question 10 On insère successivement dans l'arbre vide les entiers $1, 2, 3, 4$. Quel est l'arbre obtenu ? Est-il équilibré ? Que se passe-t-il quand on insère successivement $4, 3, 2, 1$? ◇

Solution.



□

Question 11 Montrer que ces algorithmes sont corrects. ◇

Solution. L'insertion et l'extraction utilisent toutes deux la fusion de deux arbres. On montre par induction que tous les éléments des deux arbres fusionnés se retrouvent dans le résultat de la fusion. D'autre part, la définition de la jointure préserve la propriété du tas. □

On va maintenant analyser la complexité amortie d'une suite quelconque d'insertions et d'extractions dans l'arbre. Cette analyse utilise l'idée suivante. On associe à un tas T une quantité positive $\textsc{Pot}(T)$ que l'on appelle potentiel du tas. Le potentiel du tas vide sera nul. Le temps amorti a_i de la i-ème opération est défini comme

$$a_i \stackrel{\text{def}}{=} t_i + \textsc{Pot}(T_i) - \textsc{Pot}(T_{i-1})$$

où t_i est le temps effectif qu'a pris la i-ème opération, T_{i-1} est le tas sur lequel on a effectué l'opération et T_i est le tas obtenu après la i-ème opération. Ainsi, si l'on fait la somme des temps de calcul t_i de n opérations *consécutives*, on obtient en démarrant avec un tas T_0 et en finissant avec un tas T_n :

$$\begin{aligned}\sum_{i=1}^{n} t_i &= \sum_{i=1}^{n} (a_i - \textsc{Pot}(T_i) + \textsc{Pot}(T_{i-1})) \\ &= -\textsc{Pot}(T_n) + \textsc{Pot}(T_0) + \sum_{i=1}^{n} a_i\end{aligned}$$

Si l'on montre par exemple que lorsque l'on démarre avec un tas initialement vide, le temps amorti de chaque opération est en $O(\log n)$, on montre alors que le temps total est de la forme $O(n \log n)$.

On dit qu'un nœud est bon si son sous-arbre droit est au moins aussi gros que son sous-arbre gauche et qu'il est mauvais sinon.

Question 12 On définit le potentiel $\textsc{Pot}(T)$ d'un tas T, et d'un arbre de manière plus générale, par le nombre de mauvais nœuds qu'il contient. Donner les équations récursives permettant de calculer le nombre de mauvais nœuds d'un arbre. ◇

Solution. On définit ce potentiel ainsi :

$$\begin{aligned}\textsc{Pot}(<>) &= 0 \\ \textsc{Pot}(< T_1, a, T_2 >) &= \textsc{Pot}(T_1) + \textsc{Pot}(T_2) \quad \text{si } |T_1| \leq |T_2| \\ \textsc{Pot}(< T_1, a, T_2 >) &= 1 + \textsc{Pot}(T_1) + \textsc{Pot}(T_2) \quad \text{sinon}\end{aligned}$$ □

On prend la convention que la construction d'un nouveau nœud et une comparaison d'entiers prend un temps 1 et on note $|T|$ la taille de l'arbre T et $t(op)$ le nombre d'opérations élémentaires nécessaires pour effectuer l'opération op.

Question 13 Montrer que si T et U ne sont pas tous les deux vides, alors $t(\textsc{merge}(T, U))$ est borné par :

$$2 \cdot (\log_2(|T| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U))$$

Solution. On va procéder par induction. Le cas de base correspond au cas où soit T, soit U sont vides (mais pas les deux en même temps) et dans ce cas on a $t(\textsc{merge}(T, U)) = 1$ et la borne supérieure proposée est bien valide dans ce cas.

On peut supposer sans perte de généralité que la racine de T est plus petite que la racine de U. On doit donc borner la complexité de $\textsc{join}(< T_1, a, T_2 >, U)$. On commence par utiliser l'hypothèse d'induction sur le couple (T_1, U) et l'on peut donc écrire

$$t(\textsc{merge}(T_1, U)) \leq 2(\log_2(|T_1| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T_1) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T_1, U)) \quad (9)$$

On distingue ensuite deux cas :
Le nœud a est mauvais. Dans ce cas :

$$\textsc{Pot}(T) \quad = \quad 1 + \textsc{Pot}(T_1) + \textsc{Pot}(T_2) \tag{10}$$

De plus le nœud racine de $\textsc{join}(T, U)$ est bon, et donc :

$$\textsc{Pot}(\textsc{merge}(T, U)) \quad = \quad \textsc{Pot}(T_2) + \textsc{Pot}(\textsc{merge}(T_1, U)) \tag{11}$$

De (9), l'on déduit en utilisant ensuite (11), puis l'inégalité évidente $\log_2(|T_1|) \leq \log_2(|T|)$ et enfin (10)

$$\begin{array}{rcl} t(\textsc{merge}(T_1, U)) & \leq & 2(\log_2(|T_1| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T_1) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U) + \textsc{Pot}(T_2) \\ & \leq & 2(\log_2(|T| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T_1) + \textsc{Pot}(T_2) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U)) \\ & \leq & 2(\log_2(|T| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T) - 1 + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U)) \end{array}$$

Or

$$t(\textsc{merge}(T, U)) = 1 + t(\textsc{merge}(T_1, U))$$

donc on a bien :

$$t(\textsc{merge}(T, U)) \leq 2(\log_2(|T| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U)).$$

Le nœud a est bon. Dans ce cas :

$$\begin{array}{rcl} |T_1| & \leq & |T_2| \quad \text{et donc} \\ \log_2(|T| + 1) & \geq & \log_2(2|T_1| + 2) = 1 + \log_2(|T_2| + 1) \end{array} \tag{12}$$

On a également dans ce cas :

$$\textsc{Pot}(T) \quad = \quad \textsc{Pot}(T_1) + \textsc{Pot}(T_2) \tag{13}$$

$$\textsc{Pot}(\textsc{merge}(T, U)) \quad \leq \quad 1 + \textsc{Pot}(T_2) + \textsc{Pot}(\textsc{merge}(T_1, U)) \tag{14}$$

En utilisant donc (9) puis (14), puis (12) et enfin (13)

$$\begin{array}{rcl} t(\textsc{merge}(T_1, U)) & \leq & 2(\log_2(|T_1| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T_1) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T_1, U)) \\ & \leq & 2(\log_2(|T_1| + 1) + \log_2(|U| + 1)) + \textsc{Pot}(T_1) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U)) + \textsc{Pot}(T_2) + 1 \\ & \leq & 2(\log_2(|T| + 1) + \log_2(|U| + 1)) - 1 + \textsc{Pot}(T_1) + \textsc{Pot}(T_2) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U)) \\ & \leq & 2(\log_2(|T| + 1) + \log_2(|U| + 1)) - 1 + \textsc{Pot}(T) + \textsc{Pot}(U) - \textsc{Pot}(\textsc{merge}(T, U)) \end{array}$$

Comme $t(\textsc{merge}(T, U)) = 1 + t(\textsc{merge}(T_1, U))$ on a bien l'inégalité souhaitée. □

Question 14 En déduire que la complexité d'une suite de n insertions ou extractions est en $\mathcal{O}(n \log n)$.◇

Solution. Appelons $b_1, \ldots, b_n$ la suite des éléments que l'on insère ou extrait. On définit

$$\begin{array}{rcl} T_0 & \stackrel{\text{def}}{=} & <> \\ T_i & \stackrel{\text{def}}{=} & \textsc{merge}(T_{i-1}, < b_i >) \text{ pour } i \geq 1 \text{ et une insertion} \\ T_i & \stackrel{\text{def}}{=} & \textsc{extract}(T_{i-1}) \text{ pour } i \geq 1 \text{ et une extraction} \end{array}$$

Notons t_i le temps pour effectuer l'opération l'insertion ou l'extraction conduisant au tas T_i et définissons comme expliqué précédemment le temps amorti de la i-ème opération comme :

$$a_i \stackrel{\text{def}}{=} t_i + \textsc{Pot}(T_i) - \textsc{Pot}(T_{i-1})$$

Remarquons maintenant qu'en réecrivant cette égalité quand l'opération est une insertion et en utilisant la question précédente

$$\begin{aligned} a_i &= t_i + \textsc{Pot}\left(\textsc{merge}(T_{i-1}, < b_i >)\right) - \textsc{Pot}(T_{i-1}) \\ &\leq 2\left(\log_2(|T_{i-1}| + 1) + \log_2(| < b_i > | + 1)\right) + \textsc{Pot}(< b_i >) \\ &\leq 2\left(\log_2(|T_{i-1}| + 1) + 1\right) \end{aligned}$$

Dans le cas d'une extraction, en posant $T_{i-1} =< L, b_i, R >$, on obtient

$$\begin{aligned} a_i &= t_i + \textsc{Pot}\left(\textsc{merge}(L, R)\right) - \textsc{Pot}(< L, b_i, R >) \\ &= t\left(\textsc{merge}(L, R)\right) + \textsc{Pot}\left(\textsc{merge}(L, R)\right) - \textsc{Pot}(< L, b_i, R >) \\ &\leq t\left(\textsc{merge}(L, R)\right) + \textsc{Pot}\left(\textsc{merge}(L, R)\right) - \textsc{Pot}(L) - \textsc{Pot}(R) \\ &\leq 2\left(\log_2(|L| + 1) + \log_2(|R| + 1)\right) \end{aligned}$$

On a donc dans tous les cas $\sum_{i=1}^{n} a_i = O(n \log n)$ et comme $\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} a_i + \textsc{Pot}(T_0) - \textsc{Pot}(T_n) = \sum_{i=1}^{n} a_i - \textsc{Pot}(T_n)$ on obtient $\sum_{i=1}^{n} t_i = O(n \log n)$. □

Discussion : On trouve des arguments de complexité amortie dans les structures de données que l'on redimensionne en fonction du nombre d'objets stockés, typiquement les tables de hachage ou les modules de gestion de la mémoire. L'opération de redimensionnement est faite de temps en temps et elle est relativement coûteuse (typiquement linéraire), mais ce coût est amorti sur le nombre d'opérations. Un algorithme dont la complexité amortie est satisfaisante peut malgré tout ne pas l'être dans un contexte temps réel.

Références

[1] W. Braun et M. Rem, "A logarithmic implementation of flexible arrays", Memorandum MR83/4, Eindhoven University of Technology, 1983.

[2] R. R. Hoogerwoord, "A logarithmic implementation of flexible arrays", Conference on Mathematics of Program Construction, 1992, pp. 191-207.

[3] S. Tarjan, "Self-adjusting heaps" SIAM J. Computing, 15(1), 1986, pp. 52-69.