INF431

Arbres Binaires Équilibrés
Sujet proposé par Jean-Pierre Tillich
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 ? ◇

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. ◇

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. ◇

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. ◇

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 ? ◇

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. ◇

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 ? ◇

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

Question 8 Quelle est la complexité de cette fonction ? ◇

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 $O(\log n)$. ◇

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 $4, 3, 2, 1$? ◇

Question 11 Montrer que ces algorithmes sont corrects. ◇

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. ◇

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 :

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

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

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.