Algorithmique TD6
L3 Informatique – ENS Cachan

22 octobre 2018

Exercice 1

On considère dans cet exercice des arbre binaires de recherche. En notant size(x) le nombre de nœuds d'un arbre enraciné en x, left(x) son fils gauche et right(x) son fils droit, on dit qu'un noeud est *α-équilibré* si :

$$\text{size}(\text{left}(x)) \leq \lceil \alpha(\text{size}(x) - 1) \rceil$$

et

$$\text{size}(\text{right}(x)) \leq \lceil \alpha(\text{size}(x) - 1) \rceil$$

Un arbre binaire est dit α-équilibré ssi tout nœud de l'arbre est α-équilibré.

1. Étant donné un noeud x d'un arbre, donner un algorithme pour rééquilibrer le sous-arbre enraciné en x afin d'obtenir un sous-arbre $\frac{1}{2}$-équilibré. L'algorithme devra avoir une complexité en temps et en espace linéaire en le nombre d'éléments du sous-arbre enraciné en x
2. Donner un algorithme de recherche dans un arbre binaire α-équilibré. Montrer que cet algorithme a une complexité en temps logarithmique en le nombre d'éléments de l'arbre.

On considère dans le reste du problème un $\alpha > \frac{1}{2}$. Lors de l'ajout et de la suppression d'éléments, on procède comme dans un arbre binaire de recherche habituel, sauf dans le cas ou l'on se retrouverait avec un sous-arbre qui n'est plus équilibré, auquel cas, on rééquilibre le sous-arbre maximal non équilibré avec la procédure de la question 1.

3. Implémenter l'algorithme d'insertion et de suppression d'un élément, et montrer qu'il conserve le α équilibrage d'un arbre.

On définit maintenant :

$$\Delta(x) = |\text{size}(\text{left}(x)) - \text{size}(\text{right}(x))|$$

et le potentiel d'un arbre par :

$$\Phi(t) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x)$$

où c est une constante qu'il faudra déterminer.

4. Montrer qu'un arbre $\frac{1}{2}$-équilibré possède un potentiel nul.
5. Supposons que le rééquilibrage d'un arbre à m éléments un coût en temps exactement égal à m. Démontrer et trouver une valeur de c telle que le coût amorti du rééquilibrage d'un arbre non équilibré soit en $O(1)$.
6. Montrer que la l'ajout et la suppression d'un élément dans un arbre α-équilibré a une complexité amortie en $O(\log(n))$ ou n est le nombre d'éléments de l'arbre.

Exercice 2

On se propose d'étudier les arbres splay (« Splay trees »), qui sont des arbres binaires de recherche qui "s'autoéquilibrent", en ramenant les éléments fréquemment accédés près de la racine de l'arbre. Afin de faire cela on introduit une opération, appelée « splay », qui prend un arbre A et un nœud x, et fais remonter x pour qu'il devienne la racine de l'arbre. L'opération applique récursivement des transformations locales qui ont pour but de faire remonter x petit à petit.

1. Distinguer trois cas différents (modulo symétries), et proposer une transformation locale qui fait remonter x pour chacun de ces cas, en utilisant des rotations simples ou doubles.
2. Expliquer comment implémenter les opérations suivantes en utilisant l'opération de « splay » :
   - Insérer un élément (à la fin, le nouvel élément doit être à la racine de l'arbre)
   - Supprimer un élément
   - Fusionner deux arbres splay
   - Séparer un arbre splay en deux : étant donné un nœud x de l'arbre, il faut rendre un arbre qui contient les éléments (strictement) plus petits que x, et un autre qui contient les éléments (strictement) plus grand que x.
3. On veut maintenant analyser la complexité amortie de opérations ci-dessus. On définit pour cela les notions suivantes :
   - La taille d'un nœud x : $\text{size}(x)$ est le nombre de nœuds du sous arbre qui a r pour racine.
   - Le rang d'un nœud x : $\text{rank}(x) = \log_2(\text{size}(x))$

   Trouver une fonction de potentiel telle que l'opération de « splay » au nœud x ait un coût amorti inférieur à $3(\text{rank}(r) - \text{rank}(x)) + 1$, où r est la racine de l'arbre (on comptera uniquement le nombre de rotations effectuées).
4. En déduire la complexité amortie d'une suite d'opérations quelconques.

Exercice 3

Soit Σ un alphabet fini et de cardinal supérieur ou égal à deux. On appelle codage binaire une application injective α de l'alphabet Σ dans $\{0,1\}^*$. En utilisant l'opération concaténation, α s'étend de manière naturelle en $\alpha : \Sigma^* \to \{0,1\}^*$.

Un codage est dit préfixe si aucune lettre n'est codée par un mot de code qui est préfixe du codage d'une autre lettre.

1. Montrer que pour un codage préfixe, α est injective sur Σ^*.
2. Montrer qu'on peut représenter un codage préfixe par un arbre binaire dont les feuilles sont les lettres de l'alphabet.

On dit qu'un codage est de longueur fixe quand toutes les lettres sont codées par un mot de code de même longueur. Mais on obtient des codes plus efficaces en associant des codes plus courts aux lettres qui apparaissent le plus fréquemment, quitte à devoir rallonger les codes des lettres qui apparaissent peu fréquemment.

On associe à chaque lettre a de Σ une fréquence d'apparition $f(a)$. Cette fréquence est généralement estimée à partir d'un ensemble de textes représentatif de la langue considérée.

On définit alors le coût d'un codage préfixe par la somme :

$$\sum_{a \in \Sigma} f(a) \cdot |\alpha(a)|$$

et on cherche un codage qui minimise ce coût.

3. Montrer qu'à un codage préfixe optimal correspond un arbre binaire où tout nœud interne a deux fils.

4. Montrer qu'il existe un codage préfixe optimal pour lequel les deux lettres dont le nombre d'occurences est le plus faible sont sœurs dans l'arbre.

Étant données x et y les deux lettres dont le nombre d'occurences est le plus faible dans w, on considère l'alphabet $\Sigma' = (\Sigma \setminus \{x, y\}) \cup z$ où z est une nouvelle lettre à laquelle on associe $f(z) = f(x) + f(y)$.

5. Soit T' l'arbre d'un codage optimal pour Σ', montrer que l'arbre T obtenu à partir de T' en remplaçant la feuille associée à z par un nœud interne ayant x et y comme feuilles représente un codage optimal pour Σ.
6. En déduire un algorithme recherchant un codage optimal et donner sa complexité.