ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Section d'Informatique et de Systèmes de Communication

Corrigé de la série 4 18 Octobre 2010

1. Arbres

a) On obtient les suites suivantes pour les différents parcours:

Parcours	Suite
Preorder	I, D, A, C, B, G, E, F, H, O, M, K, J, L, N
Inorder	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O
Postorder	B, C, A, F, E, H, G, D, J, L, K, N, M, O, I

b) Il s'agit de voir que pour chaque sommet r la propriété $g < r < d$ est vérifié pour tout sommet g dans le sous-arbre gauche et d dans le sous-arbre droit. Ceci est équivalent à voir que la suite obtenue par le parcours inorder est croissante (cf. point suivant). Ceci est manifestement vrai.

c) " $\Longleftarrow$ ". Nous montrons qu'un arbre de recherche résulte en une suite croissante si parcouru inorder.

Nous prouvons ce résultat par induction (forte) sur les arbres à n sommets. L'arbre vide ($n = 0$), correspond à la suite vide, qui est clairement croissante. Soit maintenant T un arbre avec racine x, sous-arbre gauche L et sous-arbre droit R. La suite obtenue par parcours inorder est

$$[\text{suite du parcours de } L], x, [\text{suite du parcours de } R].$$

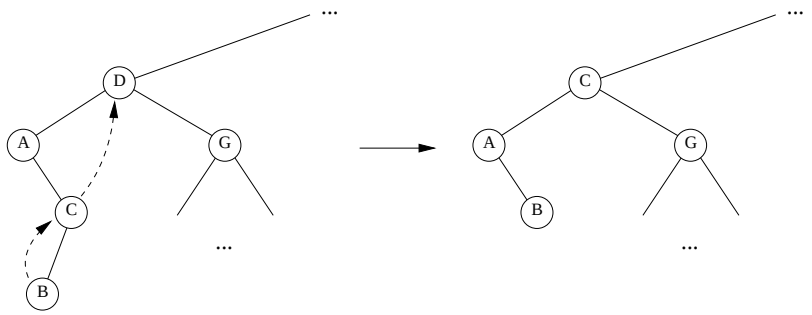
Par hypothèse d'induction, la partie L est croissante, et la partie R aussi. Il suffit donc de montrer que x est supérieur à son prédécesseur et inférieur à son succésseur. Comme T est un arbre de recherche, n'importe quel élément dans L est inférieur à x, donc aussi le dernier de la suite. De manière analogue, il n'y a pas non plus d'inversion entre x et son successeur.

" $\Longrightarrow$ ". Nous montrons que si le parcours inorder est croissant, alors l'arbre en question est un arbre de recherche. Soit x n'importe quel sommet dans l'arbre. Nous montrons que l'opération FIND(x) retourne bien le sommet x, ce qui permet de conclure.

Si x est racine de l'arbre de recherche, il sera clairement trouvé. Sinon, soit z n'importe quel sommet sur le chemin de la racine à x. Supposons sans perdre de généralité que le chemin descend dans le sous-arbre gauche de z, l'autre cas étant analogue. Alors x est parcouru avant z dans la traversée inorder, et donc $x < z$ par la croissance de la suite. Donc à l'étape où FIND se trouve en z, FIND descend aussi dans le sous-arbre gauche.

Comme cet argument s'applique à n'importe quel z sur le chemin vers x, FIND finit par trouver x.

d) On utilise l'algorithme du cours, i.e. on cherche dans le sous-arbre de gauche du sommet D le plus grand membre et on trouve C. Ce sommet ne peut pas avoir de sous-arbre droit (car ceci contredirait sa maximalité), ensuite, on remplace D par C et l'ancien sous-arbre gauche de C est mis à l'ancienne place de C. Schématiquement:



e) L'algorithme effectue plusieurs pas:

[1] Rechercher le sommet x a effacer
[2.0] S'il n'a pas de sous-arbre gauche, alors
Effacer le sommet et mettre son sous-arbre droit a la place. stop.
[2.1] Trouver le plus grand element du sous-arbre gauche de x:
$y \leftarrow \texttt{left}[x]$
Tant que y a un sous-arbre droite
$y \leftarrow \texttt{right}[y]$
[2.2] Enlever y de l'arbre et mettre son sous-arbre gauche a sa place
[2.3] Enlever x de l'arbre et mettre y a sa place.

L'algorithme fonctionne parce que le sommet y vérifie la propriété que tous les autres sommets du sous-arbre gauche sont plus petits; ceci garantit que la propriété de l'arbre de recherche est préservée. (Voir aussi page 72 des notes de cours).

f) La preuve se fait par induction sur h. Écrivons N_h le nombre maximal de sommets que peut avoir un arbre. Un arbre binaire de hauteur 0 ne peut avoir qu'un seul sommet (la racine), donc $N_0 = 1 = 2^{0+1} - 1$.

Supposons maintenant l'affirmation prouvée pour h et montrons la pour $h+1$. Notons $\mathcal{T}_h$ l'ensemble d'arbres binaires de hauteur h. Alors comme tout arbre $T \in \mathcal{T}_{h+1}$ est formé d'une racine et de deux sous-arbres $L, R \in \mathcal{T}_h$ nous avons

$$\begin{aligned} N_{h+1} &= \max_{T \in \mathcal{T}_{h+1}} \{\#(\text{sommets de } T)\} \\ &= \max_{L,R \in \mathcal{T}_h} \{1 + \#(\text{sommets de } L) + \#(\text{sommets de } R)\} \\ &= 1 + \max_{L \in \mathcal{T}_h} \{\#(\text{sommets de } L)\} + \max_{R \in \mathcal{T}_h} \{\#(\text{sommets de } R)\} \\ &= 1 + (2^{h+1} - 1) + (2^{h+1} - 1) \\ &= 2^{h+2} - 1, \end{aligned}$$

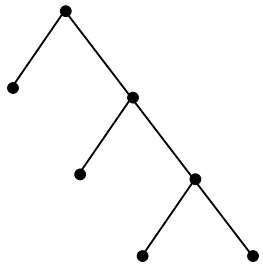
ce qui termine la preuve.

2. Graphes

a) Si la première ligne de la matrice d'adjacence de G ne contient que des 1, alors le sommet x correspondant à cette ligne est connecté à tous les autres sommets du graphe. Le graphe est donc bien connexe puisqu'il y a un chemin entre toutes paires (y, z) de sommets de G (un chemin de longueur 1 si $x \in \{y, z\}$, sinon un chemin de longueur 2 qui passe par x).

b) Attention de ne pas confondre le *degré* d'un sommet (le nombre de fils) et le *facteur d'équilibre* (la différence entre les hauteurs de ses sous-arbres).

On peut prendre par exemple le graphe ci-dessous:



Tous les sommets ont degré 0 ou 2, pourtant la racine a comme facteur d'équilibre -2.

c) • La plus grande hauteur que peut avoir T est $n-1$. On peut le montrer très facilement par induction.

• La plus petite hauteur que peut avoir T est $\lfloor \log_2 n \rfloor$. Nous avons en effet montré dans la série précédente (exercice 1f) que si T est un arbre de hauteur h avec n sommets alors

$$n \leq 2^{h+1} - 1.$$

En manipulant cette inégalité nous avons:

$$\begin{aligned} n \leq 2^{h+1} - 1 \quad &\iff \quad n + 1 \leq 2^{h+1} \\ &\iff \quad \log_2(n+1) \leq h + 1 \\ &\iff \quad h \geq \log_2(n+1) - 1, \end{aligned}$$

ainsi la plus petite hauteur que peut avoir h est

$$\lceil \log_2(n+1) - 1 \rceil .$$

On peut ensuite montrer que $\lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2(n) \rfloor$.

d) Comptons plutôt le nombre de "demi-arêtes": Chacun des n sommets est connecté a d arêtes, il y a donc nd demi-arêtes (une demi arête n'est connectée qu'à un seul sommet). Puisqu'on a 2 demi-arêtes par arête, il doit y avoir un total de $\frac{nd}{2}$ arêtes.

Donc nd doit être divisible par 2 (le nombre d'arêtes est clairement un entier). Or si n et d sont impairs alors nd sera aussi impair. On en déduit que si n est impair alors est d doit être pair.

Une autre façon (très similaire) de procéder serait de regarder la matrice d'adjacence A de G. Dans chaque ligne, exactement d entrées ont comme valeur 1 (et $(n-d)$ entrées ont

comme valeur 0). Ainsi comme il y a n lignes, le nombre total de 1's dans A est nd. Nous savons de plus les éléments sur la diagonale valent tous 0 (par hypothèse). Ces nd 1's se trouvent donc soit au dessus, soit en dessous de la diagonale.

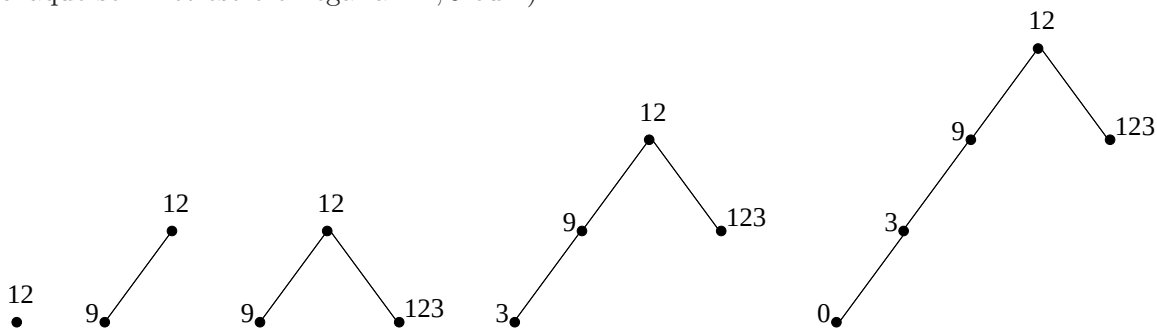
Soit x le nombre de 1's au dessus de la diagonale et y le nombre de 1's en dessous de la diagonale (donc $x + y = nd$). Puisque G est non orienté, A est symétrique et donc $x = y$. Ainsi on a

$$nd = 2x,$$

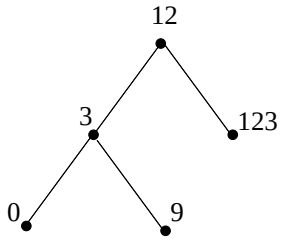
et donc nd doit être pair. Or si n et d sont impairs alors nd sera aussi impair. On en déduit que si n est impair alors est d doit être pair.

3. Arbres AVL

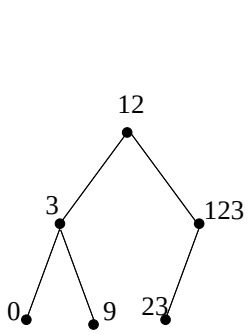
On commence par ajouter les sommets un par un, comme décrit dans le cours. Après avoir ajouté chaque sommet on vérifie que l'arbre reste AVL (c'est à dire que le facteur d'équilibre de chaque sommet est bien égal à $-1, 0$ ou 1):

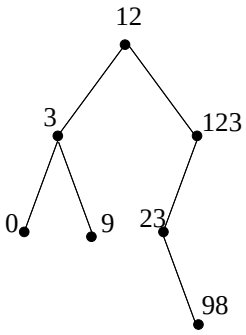


On voit à present que notre arbre n'est plus AVL puisque le facteur d'équilibre du sommet 9 vaut +2. Il faut donc effectuer une rotation pour le rééquilibrer:

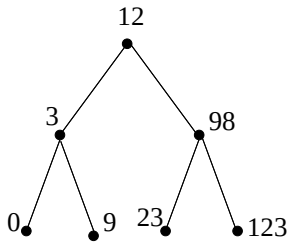


On continue à ajouter les sommets un par un, en vérifiant bien à chaque étape que notre arbre reste bien AVL:

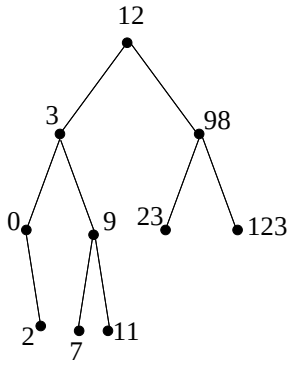




Ce dernier arbre n'est pas AVL puisque le facteur d'équilibre du sommet 123 vaut +2. On effectue de nouveau une rotation:



Finalement, on ajoute les autres sommets un par un, et on voit que l'arbre reste AVL à chaque étape, aucune rotation n'est donc ncessaire:



4. Hashing

a) Nous ne dessinons pas les tables entières ici. Pour h_1 on obtient le table de hachage suivant:

h_1	contenu
0	Alfio, Amel, Amin, Anthony, Antoine
1	Bernard, Boris
⋮	⋮
25	

Le table pour h_2 a la forme suivante:

h_2	contenu
0	
1	Jose
⋮	⋮
25	Manuel

b) La fonction h_2 est meilleure que h_1; en effet une bonne fonction de hachage doit remplir chaque entrée du tableau correspondant avec approximativement la même probabilité, afin d'éviter de longues chaînes dans les entrées du tableau. (Rappelons que la recherche dans un tableau de hachage se fait en d'abord calculant la valeur de la clé et en parcourant ensuite la liste liée de l'entrée correspondante. Le hash n'est performant que si cette liste est courte en général.)

Nous voyons que h_1 donne beaucoup de listes longues; notamment la case 9 contient les 6 entrées { Jean-Marie, Jacques, Jean-Françeois, John, Jose, Joachim }, et la case 0 contient aussi 5 entrées. En revanche, pour h_2, les listes sont en général plus courtes, le maximum dans ce cas est atteint par la case 9 qui contient 4 entrées.

c) D'une part, il est clair que h_1 ne peut pas être une très bonne fonction de hachage, parce que les premières lettres de prénoms n'ont pas du tout la même probabilité. En effet certains lettres apparaissent beaucoup plus souvent.

La fonction h_2 utilise aussi la valeur de la troisième lettre, ce qui ajoute plus de "hasard" aux adresses de hachage obtenues.