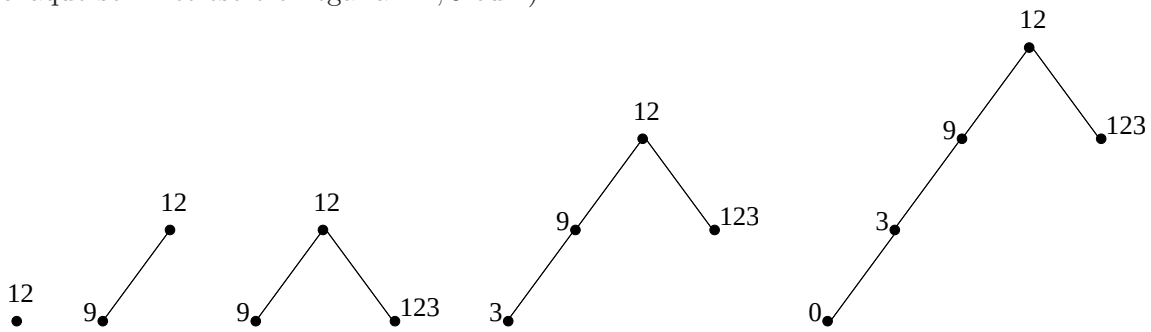
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

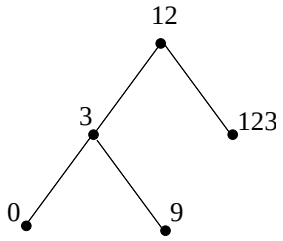
Corrigé de la série 5 19 Octobre 2007

1. Arbres AVL

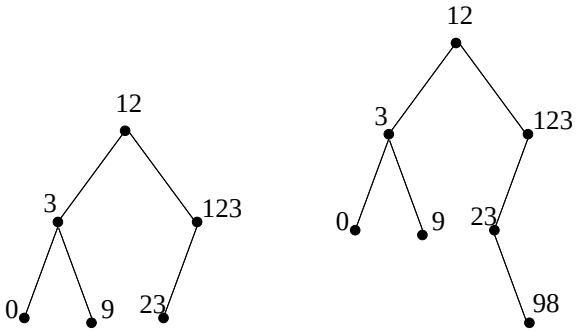
On commence par ajouter les sommets un par un, comme décrit dans le cours. Après avoir ajouté chaque sommet on vérifie que l'arbre reste AVL (c'est à dire que le facteur d'équilibre de chaque sommet est bien égal à $-1, 0$ ou 1) :



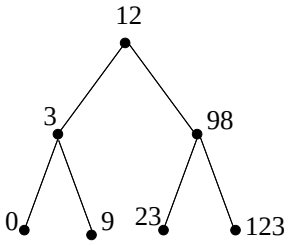
On voit à present que notre arbre n'est plus AVL puisque le facteur d'équilibre du sommet 9 vaut $+2$. Il faut donc effectuer une rotation pour le rééquilibrer :



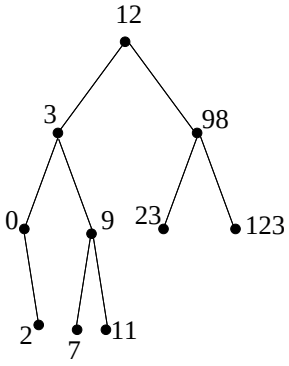
On continue à ajouter les sommets un par un, en vérifiant bien à chaque étape que notre arbre reste bien AVL :



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 :



2. 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-François, 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.

3. Algorithmes récursifs : Le diamètre d'un arbre binaire

a) Notons r la racine de T. Pour un chemin dans T il y a trois possibilités :

(1) Le chemin se trouve entièrement dans T_1,

(2) Le chemin se trouve entièrement dans T_2,

(3) Le chemin passe par (ou se termine à) la racine r.

Pour pouvoir dire quelque chose sur le diamètre de T, nous considérons un chemin γ de longueur maximale. Si γ est du premier type, alors la longueur de γ est certainement égale à $\mathrm{diam}(T_1)$, et si γ est du deuxième type, sa longueur est égale à $\mathrm{diam}(T_2)$.

Le troisième cas nécessite une discussion un peu plus détaillée. Dénotons le dernier élément du chemin dans T_1 par x_1 et le dernier élément dans T_2 par x_2. Le chemin est alors comme suit :

$$x_1 \to \cdots \to \text{ racine de } T_1 \ \to \ \text{racine de } T \ \to \ \text{racine de } T_2 \ \to \cdots \to x_2.$$

La longueur de ce chemin est donc

$$\mathrm{profondeur}_{T_1}(x_1) + 1 + 1 + \mathrm{profondeur}_{T_2}(x_2),$$

où profondeur$_T(x)$ dénote la profondeur du sommet x dans l'arbre T (voir p.56 du cours pour voir la définition de profondeur).

La maximalité du chemin implique maintenant que les deux termes profondeur$_{T_1}(x_1)$ et profondeur$_{T_2}(x_2)$ doivent être maximaux, i.e. que les sommets en question se trouvent en tout en bas de l'arbre et donc que la profondeur des sommets en question est égale à la hauteur du sous-arbre correspondant. En résumé donc, la longueur du chemin est, dans ce cas, égale à

$$h_1 + 2 + h_2,$$

o h_i dénote la hauteur de l'arbre T_i.

Nous déduisons donc que la longueur du plus long chemin dans T (et alors le diamètre de T) est égale à

$$\max\{\text{diam}(T_1), \text{diam}(T_2), h_1 + h_2 + 2\}.$$

Soit h la hauteur de T. Il est alors assez clair que

$$h = \max(h_1, h_2) + 1.$$

b) Nous construisons une fonction récursive qui calcule à la fois la hauteur et le diamètre d'un arbre binaire étant donné sa racine :

Call : HauteurEtDiam
Input: r : Racine d'un arbre binaire.
Output: (h, d) : Hauteur et diamètre de l'arbre.

```
if r = 0 then
    return (−1, −1)
(h1, d1) ← HauteurEtDiam(r[left])
(h2, d2) ← HauteurEtDiam(r[right])
h ← max{h1, h2} + 1
d ← max{d1, d2, h1 + h2 + 2}
return (h, d)
```

Remarquons que cet algorithme a bien un temps de parcours linéaire : Si nous ignorons pour un instant les appels récursifs, nous voyons que, puisqu'il n'y a pas de boucles, l'algorithme opère en temps constant. Mais l'appel de la fonction HauteurEtDiam se fait exactement une fois pour chaque sommet ; il en résulte que l'algorithme est $O(|V|)$.

4. Les tours de Hanoi

a) Dans ce qui suit, ThirdOf est la fonction qui associe retourne pour deux nombres entre 1 et 3 donnés, le troisième nombre (par exemple ThirdOf$(3, 1) = 2$).

Nous pouvons définir la fonction qui déplace une tour de taille n récursivement comme suit :

Call : HanoiTowerMove
Input: n, s, d, n : Taille de la tour, bâtons source et destination
Output: Une suite d'appels à MoveSlice.

```
if n = 1 then
```

MoveSlice(s, d).
return
$t \leftarrow$ ThirdOf(s, d)
HanoiTowerMove$(n-1, s, t)$
MoveSlice(s, d)
HanoiTowerMove$(n-1, t, d)$

b) Notons T_n le nombre de MoveSlice nécessaire pour déplacer une tour de taille n. Nous avons clairement $T_1 = 1$. La définition récursive ci-dessus permet aussi de déduire la formule inductive

$$T_n = 2T_{n-1} + 1.$$

Nous prouvons maintenant par induction la formule fermée $T_n = 2^n - 1$. Elle est clairement vraie pour $n = 1$. Pour $n > 1$, nous concluons en appliquant l'hypothèse d'induction au cas $n - 1$ que

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ &= 2 \cdot (2^{n-1} - 1) + 1 \\ &= 2^n - 1. \end{aligned}$$

Ceci termine la preuve.

Remarquons que T_n croît exponentiellement vite, ce qui implique que même pour n de taille modérée, cet algorithme ne terminera pas en un temps raisonnable.

c) Notons $\tilde{T}_n$ le temps utilisé pour déplacer une tour de taille n en utilisant une méthode optimale quelconque. Il nous faut alors montrer que pour tout n on a $\tilde{T}_n \geq T_n$. Clairement, pour $n = 1$ on ne peut pas faire mieux qu'en une étape, et donc $\tilde{T}_1 \geq 1 = T_1$. Nous montrons maintenant qu'on a

$$\tilde{T}_n \geq 2\tilde{T}_{n-1} + 1,$$

pour tout $n > 1$, ce qui nous permettra de conclure.

Supposons que nous déplaçons une tour de taille n avec la méthode optimale. Il existe donc une étape m, $1 \leq m \leq \tilde{T}_n$ qui déplace le plus grand disque pour la dernière fois.

En l'étape m, nous avons :

(i) Le n-ème disque est placé sur le bâton but, et aucun des $n-1$ disques plus petits peut déjà se trouver à cet endroit (parce que le n-ème disque est plus grand que les $n-1$ autres disques de la tour).

(ii) Le bâton duquel le n-ème disque est pris contient, à cet instant, aucun des $n-1$ plus petits disques de la tour, parce que le n-ème disque est enlevé d'en haut. Il s'en suit que les $n-1$ plus petits disques forment une tour sur le troisième bâton.

Nous en déduisons déjà qu'après le m-ème mouvement nous avons encore besoin de déplacer la tour à $n-1$ disques au bon endroit, ce qui nécessite donc au moins $\tilde{T}_{n-1}$ étapes.

Avec des raisonnements analogues, si l'on considère l'étape m' du *premier* déplacement du n-ème disque, nous voyons que *avant* l'étape m' il nous a fallu déplacer une tour de taille $n-1$, ce qui ajoute encore une fois $\tilde{T}_{n-1}$ opérations.

Nous avons donc que $\tilde{T}_n \geq 2\tilde{T}_{n-1} + 1$, ou le $+1$ vient de l'étape m (nous ne comptons pas l'étape m', parce qu'il se peut que $m = m'$).

Le fait que la fonction $x \mapsto 2x + 1$ est croissante permet maintenant de conclure par induction que

$$\tilde{T}_n \geq T_n$$

pour tout n.