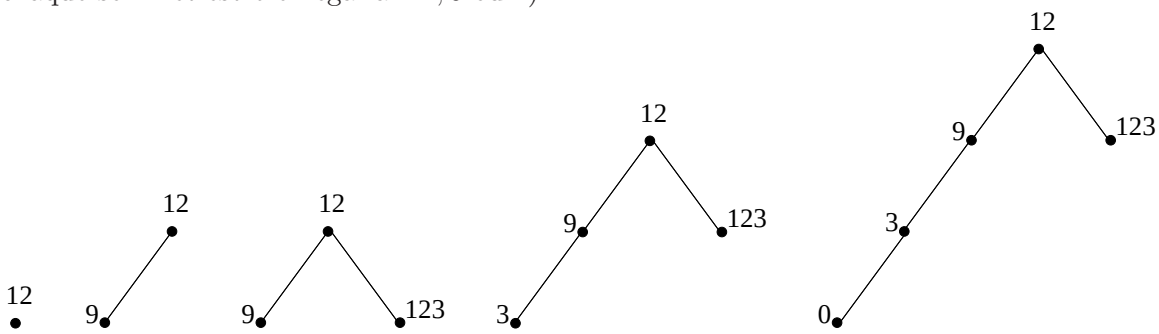
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

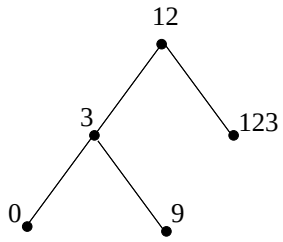
Corrigé de la série 5 19 Octobre 2009

1. Arbres AVL

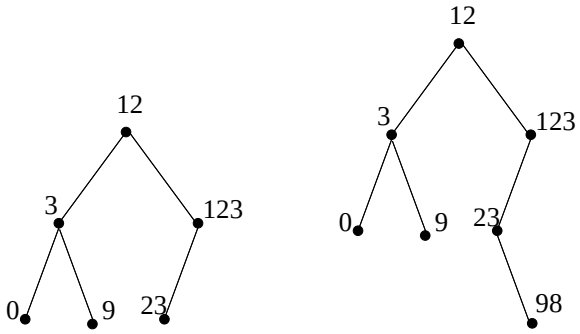
On commence par ajouter les sommets un par un, comme décrit dans le cours. Après avoir ajouté chaque sommet on vérifie que l'arbre reste AVL (c'est à dire que le facteur d'équilibre de chaque sommet est bien égal à $-1, 0$ ou 1):



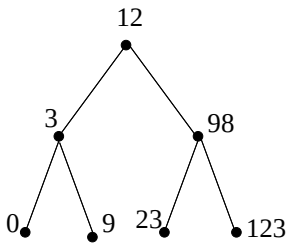
On voit à present que notre arbre n'est plus AVL puisque le facteur d'équilibre du sommet 9 vaut $+2$. Il faut donc effectuer une rotation pour le rééquilibrer:



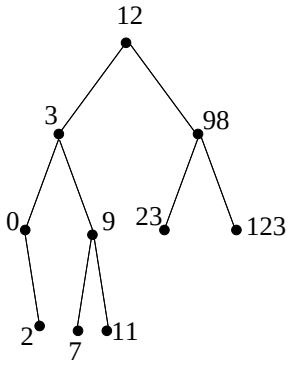
On continue à ajouter les sommets un par un, en vérifiant bien à chaque étape que notre arbre reste bien AVL:



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 suivante:

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. Diviser pour régner: Le problème du skyline

a) Nous parcourons l'axe des x de gauche à droite, en regardant les points un par un. Pour chaque point de L et de R nous devons décider s'il faut ajouter un point C, et si oui quelle va être sa hauteur.

Un algorithme est donnée ci-dessous. On remarque que le nombre d'opérations est $O(|L|+|R|)$ où $|L|$ et $|R|$ représentent les nombres de points dans les skylines de L et R.

Dans l'algorithme ci-dessous, les skylines sont données sous forme de tableau, avec les entrées étant des couples (x, h). Par convention, la dernière entrée est toujours $(\infty, 0)$ et marque la fin de la skyline.

Call: SkylineMerge
Input: Skylines L et R
Output: Skyline ret, superposition des deux skylines.

```
i_ℓ, i_r ← 0
h_ℓ ← 0, h_r ← 0, h ← 0
while min(L[i_ℓ].x, R[i_r].x) < ∞ do
  if L[i_ℓ].x < R[i_r].x then
    h_ℓ ← L[i_ℓ].h
    x ← L[i_ℓ].x
    i_ℓ ← i_ℓ + 1
  else
    h_r ← L[i_r].h
    x ← L[i_r].x
    i_r ← i_r + 1
  if max(h_ℓ, h_r) ≠ h then
    h ← max(h_ℓ, h_r)
    ret.add((x, h))
ret.add((∞, 0))
return ret
```

Dans le pseudocode ci-dessus, la méthode add ajoute un élément à la fin d'un array.

b) Remarquons d'abord que l'algorithme SkylineMerge ne résout pas le problème Skyline. En effet, Skyline attend comme input un ensemble de bâtiments (i.e. de triplets (x_1, x_2, h)) et retourne comme output le skyline correspondant. Par contre, SkylineMerge a comme input deux skylines et retourne le skyline obtenu en les combinant.

MergeSort: Notre solution est très similaire à *MergeSort*, nous rappelons donc les détails de cet algorithme. Le problème à résoudre est d'ordonner une suite de nombres naturels (l'input est donc une suite de nombres naturels, et l'output une suite qui contient les mêmes éléments, mais ordonnés). Nous avons d'abord construit un algorithme Merge qui, étant donné deux suites S_1 et S_2 de nombres naturels *ordonnées*, retourne une suite ordonnée contenant les éléments de S_1 et S_2.

L'algorithme MergeSort divise la suite donnée en 2 sous-suites (plus petites), qui sont ensuite ordonnées (avec un appel récursif à MergeSort). Puis, ces 2 sous-suites ordonnées sont combinées avec l'algorithme Merge pour obtenir une version ordonnée de la suite de l'input.

Puisque nous avons fait un appel récursif à MergeSort, chacune des sous-suites sera elle même divisée en deux sous-suites, etc. Il faut donc définir un cas de base où la récurrence se terminera. Il s'agit du cas où la suite ne contient qu'un seul élément et est donc forcément déjà ordonnée.

On l'appelle un algorithme diviser-pour-régner puisqu'on divise le problème en deux instances plus petites du même problème. l'algorithme Merge est utilisé pour recoller les solutions de ces instances pour obtenir la solution du problème original.

Pour résoudre le problème Skyline nous procédons de la même façon. Notre algorithme (que nous appelons SkylineSolve) commence par diviser la liste de bâtiments en deux sous-listes (dont la taille vaut à peu près la moitié de celle de la suite originale). Nous

trouvons ensuite le Skyline de chacune de ces suites avec un appel récursif à SkylineSolve.

Nous réduisons donc notre problème à deux sous-problèmes qui sont toujours des instances de Skyline, mais avec un input plus petit (il y a moins de bâtiments).

Les solutions de ces deux sous-problèmes (les deux skylines) sont ensuite combinées avec l'algorithme SkylineMerge donné dans la partie a) (ce pas correspond donc à Merge).

Comme avec MergeSort, puisqu'il y a cet appel récursif, il faut définir un cas de base. C'est le cas où il n'y a qu'un seul bâtiment, et le skyline est égal a ce bâtiment.

On obtient au final, en pseudocode:

Call: SkylineSolve
Input: Bat: Une liste de n bâtiments (de triplets (x_1, x_2, h))
Output: A: Le skyline correspondant (une suite de points (x, h))
  if $n = 1$ **then**
    return $((Bat[1].x_1, Bat[1].h), (Bat[1].x_2, 0))$
  else
    $m \leftarrow \lceil n/2 \rceil$
    $Bat_1 \leftarrow$ la sous-suite de Bat composée des $m-1$ premiers éléments de Bat
    $Bat_2 \leftarrow$ la sous-suite de Bat composée des autres éléments (bâtiments) de Bat
    $Skyline_1 \leftarrow$ SkylineSolve(Bat_1)
    $Skyline_2 \leftarrow$ SkylineSolve(Bat_2)
    return SkylineMerge$(Skyline_1, Skyline_2)$