Corrigé

Examen d'algorithmique

EPITA ING1 2012 S1; A. DURET-LUTZ

Durée : 1 heure 30

Janvier 2010

Consignes

- Cet examen se déroule **sans document** et **sans calculatrice**.
- Répondez sur le sujet dans les cadres prévus à cet effet.
- Il y a six pages d'énoncé, et une page d'annexe.
  Rappelez votre nom en haut de chaque feuille au cas où elles se mélangeraient.
- Ne donnez pas trop de détails. Lorsqu'on vous demande des algorithmes, on se moque des points-virgules de fin de commande etc. Écrivez simplement et lisiblement. Des spécifications claires et implémentables sont préférées à du code C ou C++ verbeux.
- Le barème est indicatif et correspond à une note sur 25.

1 Notation Θ (3 points)

1. **(2 points)** Prouvez rigoureusement que quelles que soient trois constantes a, b, c strictements positives, on a $a\log(bn + c) = \Theta(\log(n))$

Réponse :

Commençons par un raisonnement que beaucoup ont fait, et qui est malheureusement faux : $bn + c = \Theta(n)$ **donc** $\log(bn + c) = \Theta(\log(n))$. Ceci suppose, à tort, que $f(\Theta(g(n))) = \Theta(f(g(n)))$. Nous avons vu en cours un cas typique où la composition ne marche pas : $2n = \Theta(n)$ mais pour autant $4^n = 2^{2n} \neq \Theta(2^n)$. Montrer que $\log(bn + c) = \Theta(\log(n))$ demande donc une preuve, ce qu'on vous demandait ici.

Deux méthodes étaient envisageables.

La première s'appuie sur le fait que $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^{+\star} \iff f(n) \in \Theta(g(n))$. Attention, dans cette propriété il faut que c soit non nulle. Beaucoup d'entre vous se sont malheureusement contentés de montrer $c \in \mathbb{R}$.

$$\begin{aligned}\lim_{n\to\infty} \frac{a\log(bn+c)}{\log n} &= a \lim_{n\to\infty} \frac{\log(n(b+\frac{c}{n}))}{\log n} = a \lim_{n\to\infty} \left(\frac{\log n}{\log n} + \frac{\log(b+\frac{c}{n})}{\log n}\right) \\ &= a\left(1 + \lim_{n\to\infty} \frac{\log(b+\frac{c}{n})}{\log n}\right) = a\left(1 + \lim_{n\to\infty} \frac{\log b}{\log n}\right) = a(1+0) = a \neq 0.\end{aligned}$$

Une autre méthode, plus ardue, consiste à utiliser la définition de Θ. C'est-à-dire qu'on doit montrer qu'il existe (ce qui revient souvent à trouver) $n_0 \geq 0$, $c_1 > 0$ et $c_2 > 0$ tels que $\forall n \geq n_0$, $0 \leq c_1 log(n) \leq a\log(bn + c) \leq c_2 \log(n)$.

Attention, il est **faux** de chercher à minorer $a\log(bn + c)$ ainsi :

$$\forall n \geq \underbrace{0}_{n_0}, a\log(bn + c) \geq a\log(bn) = a\log b + a\log n \geq \underbrace{a}_{c_1}\log n$$

L'erreur se trouve dans la dernière inégalité : en effet $a\log b$ n'est pas forcément positif, il suffit que b ait le mauvais goût d'être compris entre 0 et 1 (strictement). L'astuce consiste à réaliser que même si $a\log b$ peut être négatif, il s'agit tout de même d'une constante dont l'effet sur $a\log n$ sera de moins en moins important lorsque n croît, et n'influera pas sur le comportement asymptotique. On peut donc trouver un n_0 à partir duquel on a par exemple $a\log b + a\log n \geq \frac{a}{2}\log n$.

On démontre ainsi d'une part que pour tout $n \geq \frac{1}{b^2}$ on a $\frac{a}{2}\log(n) \geq -a\log(b)$ donc

$$a\log(bn+c) \geq a\log(bn) = a\log(b) + a\log(n) \geq a\log(b) + \frac{a}{2}\log(n) - a\log(b) = \underbrace{\frac{a}{2}}_{c_1}\log(n)$$

et d'autre part, pour tout $n \geq \max(c, b+1)$

$$a\log(bn+c) \leq a\log(bn+n) = a(\log(b+1) + \log(n)) \leq a(\log(n) + \log(n)) = \underbrace{2a}_{c_2}\log(n)$$

Il donc existe c_1 et c_2 constantes strictement positives, telles que

$$n \geq \underbrace{\max(1/b^2, c, b+1)}_{n_0}, \quad c_1\log(n) \leq a\log(bn+c) \leq c_2\log(n)$$

on en déduit $a\log(bn + c) = \Theta(\log(n))$

2. **(1 point)** Laquelle ou lesquelles des constantes a, b et c peuvent être prises nulles sans invalider l'égalité ci-dessus ?

Réponse :

c peut être nulle. Les autres apparaissent dans les définitions de c_1 et c_2 qui n'ont pas le droit d'être nulles.

2 Programmation Dynamique (13 points)

Soient $m_1 < m_2 < \cdots < m_n$ des entiers stockés dans un arbre binaire de recherche. Chaque entier m_i est associé à un poids w_i qui représente la fréquence avec lequel il va être recherché dans l'arbre (plus w_i est grand, plus m_i est recherché souvent par le programme). Notre objectif va être de construire l'arbre de recherche le plus efficace en fonction de ces poids qui sont connus à l'avance.

Pour formaliser la notion d'arbre efficace, définissons le coût d'accès pondéré C d'un arbre binaire de recherche pour n entiers comme

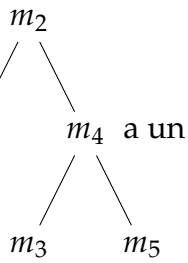
$$C = \sum_{i=1}^{n} (d_i + 1) w_i$$

où d_i représente la profondeur du nœud représentant la valeur m_i dans l'arbre (la racine étant à la profondeur 0). Intuitivement $d_i + 1$ correspond au nombre de comparaisons à faire avant de trouver m_i dans l'arbre.

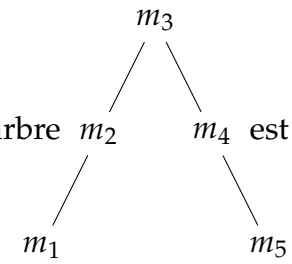
Par exemple considérons deux arbres binaires de recherches pour les valeurs

i	1	2	3	4	5
m_i	2	5	6	8	9
w_i	10	3	15	2	7

.



L'arbre a un coût d'accès pondéré de $C = 2w_1 + 1w_2 + 3w_3 + 2w_4 + 3w_5 = 93$.



Tandis que le coût d'accès pondéré de l'arbre est $C = 3w_1 + 2w_2 + 1w_3 + 2w_4 + 3w_5 = 76$.

Si l'on ne devait choisir qu'entre ces deux arbres, on garderait le second, de coût plus faible. On veut (malheureusement pour vous) trouver l'arbre de coût minimal parmi **tous** les arbres binaires de recherche possibles pour les valeurs $m_1, \ldots, m_n$.

Il est possible de trouver cet arbre par programmation dynamique.

Notons $C(i,j)$ le coût d'accès pondéré **minimal** des arbres binaires de recherche pour les valeurs $m_i, \ldots, m_j$. On a

$$C(i,j) = \begin{cases} 0 & \text{si } i > j \\ w_i & \text{si } i = j \\ \min\limits_{k \in [\![i,j]\!]} \left(C(i,k-1) + C(k+1,j) + \sum\limits_{m=i}^{j} w_m \right) & \text{si } i < j \end{cases}$$

1. **(2 points)** Justifiez la dernière ligne (cas $i < j$) de la définition récursive de $C(i,j)$.

Réponse :

Pour créér l'arbre de recherche de coût minimal pour $m_i, \ldots, m_j$ on choisit un m_k comme racine, puis on calcule des arbres de coût minimaux pour $m_i, \ldots, m_{k-1}$ à gauche, et $m_{k+1}, \ldots, m_j$ à droite. Le coût de l'arbre ainsi construit est celui du sous-arbre gauche $C(i, k-1) + \sum_{m=i}^{k-1} w_m$ (la somme ajoutée est là parce que l'arbre est un niveau plus bas que ce que la définition de C considère) plus celui du sous-arbre droit $C(k+1, j) + \sum_{m=k+1}^{j} w_m$ plus le poids de la racine, w_k. Cela donne bien la somme $C(i, k-1) + C(k+1, j) + \sum_{m=i}^{j} w_m$, que l'on cherche à minimiser par rapport à tous les choix de racine (k) possibles.

2. **(4 points)** Completez le tableau des $C(i, j)$ suivant pour les valeurs de l'exemple.

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	10	16	44	48	64
$i=2$	0	3	21	25	41
$i=3$	0	0	15	19	35
$i=4$	0	0	0	2	11
$i=5$	0	0	0	0	7

3. **(2 points)** Pour n valeurs dans l'arbre, quelle est la complexité de l'algorithme qui remplit ce tableau (justifiez votre réponse sans écrire l'algorithme).

Réponse :

Le tableau comporte $n \times n$ cases, chaque case demande au pire n opérations (pour le calcul du min), donc la complexité est dans $O(n^3)$.

Un décompte plus précis montre que cette borne est atteinte. En effet :

- les n cases de la première diagonale ($i = j$) demandent chacune 1 seule opération
- les $n-1$ cases de la seconde diagonale ($i = j-1$) demandent chacune 2 calculs dans le min
- ...
- les $n-k$ cases de la $(k+1)^{\text{e}}$ diagonale ($i = j-k$) demandent chacune $k+1$ calculs
- ...
- l'unique case de la n^{e} diagonale ($i = j-(n-1)$) demandent chacune n calculs

Le nombre d'opération nécessaires pour remplir le triangle suppérieur droit du tableau ($i \leq j$) est donc

$$\sum_{k=n}^{1} k(n-k) = n\sum_{k=1}^{n} k - \sum_{k=1}^{n} k^2 = \frac{n^2(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} = \frac{(n-1)n(n+1)}{6} = \Theta(n^3)$$

À cela il faut ajouter les $\Theta(n^2)$ opérations pour mettre des 0 dans les autres cases. La complexité est donc exactement $\Theta(n^3)$.

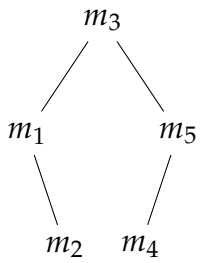
4. **(3 points)** La personne qui a écrit l'algorithme avait suivi les cours d'algo, donc elle a aussi pris la peine de sauvegarder dans un tableau séparé la valeur du k qui donnait le coût minimal dans le calcul de $C(i, j)$. Le tableau obtenu est le suivant :

	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	1	3	3	3
$i=2$		3	3	3
$i=3$			3	3
$i=4$				5

Déduisez-en et dessinez l'arbre binaire de recherche de coût d'accès pondéré minimal pour les valeurs de l'exemple.

Réponse :

Si on appelle K le tableau ci-dessus, $K(1,5) = 3$ indique que m_3 est la racine de l'arbre. $K(1,2) = 1$ indique que m_1 est le fils gauche de m_3 et $K(4,5) = 5$ indique que m_5 est le fils droit m_1. Cela nous donne l'arbre suivant :



On peut vérifier que son coût minimal $C = 2w_1 + 3w_2 + w_3 + 3w_4 + 2w_5 = 64$ est bien ce qu'on retrouve dans le tableau en $C(1,5)$.
Note : c'est ici l'unique ABR avec ce poids. Beaucoup d'entre vous ont produit d'autres arbres binaires qui n'étaient pas des arbres binaires de recherche.

5. **(2 points)** Pour 5 valeurs distinctes, comme dans l'exemple, combien peut-on créer d'arbres binaires de recherche différents ? (Sans prendre en compte les poids. C'est juste du dénombrement.)

Réponse :

Soit $C(n)$ le nombre "formes" d'arbres binaires de n nœuds. (Le fait que ce soit des arbres de recherche ne change rien à l'affaire.) On a $C(1) = 1$, $C(2) = 2$. Dans le cas général, si on décide que la racine est le k^{e} nœud, il y a $C(k-1)C(n+1-k)$ arbres possibles (si l'on convient que $C(0) = 1$). Il faut ensuite sommer cela pour tous les k possibles :

$$C(n) = \sum_{k=1}^{n} C(k-1)C(n-k)$$

Cela nous donne

$$\begin{aligned} C(3) =& C(0)C(2) + C(1)C(1) + C(2)C(0) = 5 \\ C(4) =& C(0)C(3) + C(1)C(2) + C(2)C(1) + C(3)C(0) = 14 \\ C(5) =& C(0)C(4) + C(1)C(3) + C(2)C(2) + C(3)C(1) + C(4)C(0) = 42 \end{aligned}$$

La réponse est donc 42.
Le problème est similaire à celui du dénombrement des parenthésages dans le problème des chaînes de multiplication de matrices : le nombre de parenthésages $P(n)$ est en fait égal à $C(n+1)$. $C(n)$ est le n^{e} nombre de Catalan.

3 File de priorité (4 points)

Dans cette question on considère une file de priorité S implémentée à l'aide d'un tas « max », c'est-à-dire avec la valeur maximale à la racine du tas.

S est initialement vide. Donnez l'état du tableau représentant S après y avoir effectué chacune des opérations suivantes :

– `insert(5)`

5

- `insert(2)`

5	2

- `insert(7)`

7	2	5

- `insert(6)`

7	6	5	2

- `insert(4)`

7	6	5	2	4

- `extract_max()`

6	4	5	2

- `extract_max()`

5	4	2

- `insert(6)`

6	5	2	4

- `extract_max()`

5	4	2

- `extract_max()`

4	2

4 Recherche par interpolation (5 points)

Voici un algorithme de recherche dans un tableau d'entiers. Le principe est similaire à la recherche dichotomique, sauf qu'au lieu de sonder le tableau *en son milieu* avant d'explorer l'un des côtés, le point de coupe est choisi par interpolation en fonction des valeurs des extrémités et de la valeur recherchée.

Entrées :
A : un tableau d'entiers, trié ;
l : le plus petit indice du tableau ;
r : le plus grand indice du tableau ;
v : la valeur à rechercher parmi $A[l..r]$.
Sortie :
l'indice de v dans A s'il existe, 0 sinon.

```
INTERPOLATIONSEARCH(A, l, r, v)
 1  if l ≤ r then
 2      m ← ⌊ (r − l)/(A[r] − A[l]) (v − A[l]) + l ⌋
 3      if v = A[m] then
 4          return m
 5      else
 6          if v < A[m] then
 7              return INTERPOLATIONSEARCH(A, l, m − 1, v)
 8          else
 9              return INTERPOLATIONSEARCH(A, m + 1, r, v)
10  else
11      return 0
```

1. **(1 point)** Les appels récursifs de cet algorithme sont-ils terminaux ?

Réponse :

Oui, dans les deux cas l'appel récursif est la dernière opération de la fonction.
Note : la question n'était pas de s'avoir si l'algorithme terminait.

2. **(2 point)** Donnez une définition *récursive* de la complexité en pire cas de cet algorithme, en fonction de la taille du tableau $n = r - l + 1$.

Réponse :

Au pire, le tableau dans lequel la récursion est effectuée possède $n - 1$ éléments. On a donc :

$$T(n) = T(n-1) + \Theta(1)$$

3. **(1 point)** Quelle est la solution de cette équation ?

Réponse :

Dans le pire cas, on a donc

$$T(n) = \Theta(n)$$

4. **(1 point)** Quelle est la complexité de l'algorithme dans le meilleur cas ?

Réponse :

Dans le meilleur cas l'algorithme tombe sur la valeur recherchée dès le premier test. On a donc $T(n) = \Theta(1)$ dans le meilleur cas.

Notations asymptotiques

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$
$$\Omega(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$
$$\Theta(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^{+\star}, \exists c_2 \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff g(n) \in \mathrm{O}(f(n))$ et $f(n) \notin \mathrm{O}(g(n))$ $\quad f(n) \in \mathrm{O}(g(n)) \iff g(n) \in \Omega(f(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f(n) \in \mathrm{O}(g(n))$ et $g(n) \notin \mathrm{O}(f(n))$ $\quad f(n) \in \Theta(g(n)) \iff \begin{cases} f(n) \in \Omega(g(n)) \\ g(n) \in \Omega(f(n)) \end{cases}$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^{+\star} \iff f(n) \in \Theta(g(n))$

Ordres de grandeurs

constante	$\Theta(1)$	
logarithmique	$\Theta(\log n)$	
polylogarith.	$\Theta((\log n)^c)$	$c > 1$
	$\Theta(\sqrt{n})$	
linéaire	$\Theta(n)$	
	$\Theta(n \log n)$	
quadratique	$\Theta(n^2)$	
	$\Theta(n^c)$	$c > 2$
exponentielle	$\Theta(c^n)$	$c > 1$
factorielle	$\Theta(n!)$	
	$\Theta(n^n)$	

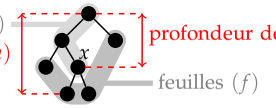
Théorème général

Soit à résoudre $T(n) = aT(n/b) + f(n)$ avec $a \geq 1$, $b > 1$

- Si $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ pour un $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.
- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour un $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour un $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$.

(Note : il est possible de n'être dans aucun de ces trois cas.)

Arbres



Pour tout arbre binaire :

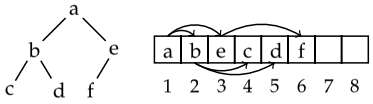
$n \leq 2^{h+1} - 1$ $\quad h \geq \lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$ si $n > 0$

$f \leq 2^h$ $\quad h \geq \lceil \log_2 f \rceil$ si $f > 0$

$f = ni + 1$ (si l'arbre est *complet* = les nœud internes ont tous 2 fils)

Dans un arbre binaire équilibré une feuille est soit à la profondeur $\lfloor \log_2(n+1) - 1 \rfloor$ soit à la profondeur $\lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$. Pour ces arbres $h = \lfloor \log_2 n \rfloor$.

Un arbre parfait (= complet, équilibré, avec toutes les feuilles du dernier niveau à gauche) **étiqueté** peut être représenté par un tableau.



Les indices sont reliés par :

$\text{Père}(y) = \lfloor y/2 \rfloor$

$\text{FilsG}(y) = y \times 2$

$\text{FilsD}(y) = y \times 2 + 1$

Identités utiles

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \quad \text{si } x \neq 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{si } |x| < 1$$

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad \text{si } |x| < 1$$

$$\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$$

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Définitions diverses

La complexité d'un problème est celle de l'algorithme le plus efficace pour le résoudre.

Un tri stable préserve l'ordre relatif de deux éléments égaux (au sens de la relation de comparaison utilisée pour le tri).

Un tri en place utilise une mémoire temporaire de taille constante (indépendante de n).

Rappels de probabilités

Espérance d'une variable aléatoire X **:** C'est sa valeur attendue, ou moyenne. $\mathrm{E}[X] = \sum_x \Pr\{X = x\}$

Variance : $\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - \mathrm{E}^2[X]$

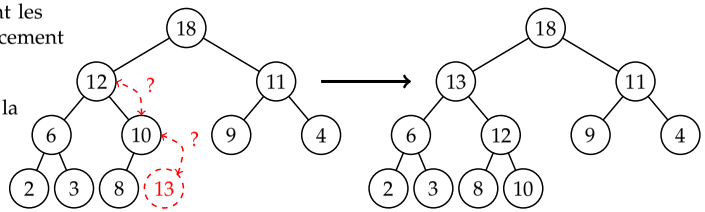
Loi binomiale : On lance n ballons dans r paniers. Les chutes dans les paniers sont équiprobables ($p = 1/r$). On note X_i le nombre de ballons dans le panier i. On a $\Pr\{X_i = k\} = C_n^k p^k (1-p)^{n-k}$. On peut montrer $E[X_i] = np$ et $\mathrm{Var}[X_i] = np(1-p)$.

Tas

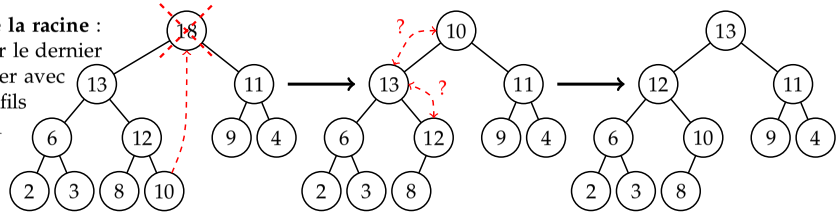
Un tas est un arbre parfait partiellement ordonné : l'étiquette d'un nœud est supérieure à celles de ses fils.

Dans les opérations qui suivent les arbres parfaits sont plus efficacement représentés par des tableaux.

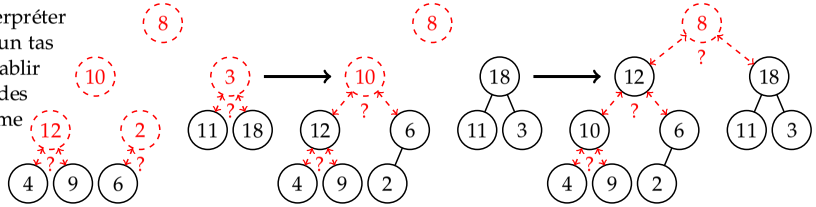
Insertion : ajouter l'élément à la fin du tas, l'échanger avec son père tant qu'il lui est inférieur (en remontant vers la racine). $T_{\text{insert}} = \mathrm{O}(\log n)$



Suppression de la racine : la remplacer par le dernier nœud ; l'échanger avec son plus grand fils tant que celui-ci est plus grand. $T_{\text{rem}} = \mathrm{O}(\log n)$



Construction : interpréter le tableau comme un tas (incorrect) puis rétablir l'ordre en partant des feuilles (vues comme des tas corrects). $T_{\text{build}} = \Theta(n)$



Arbres Rouge et Noir

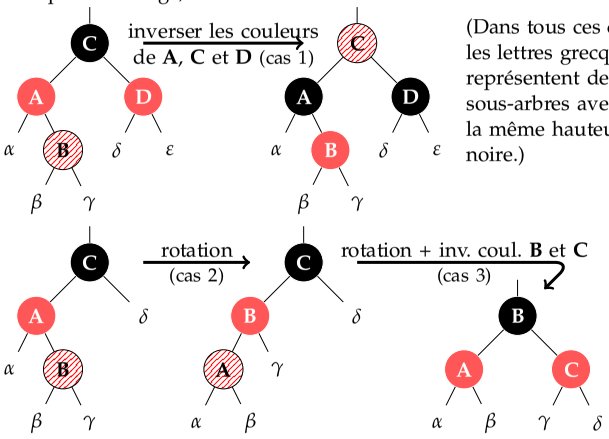
Les ARN sont des arbres binaires de recherche dans lesquels : (1) un nœud est rouge ou noir, (2) racine et feuilles (NIL) sont noires, (3) les fils d'un nœud rouge sont noirs, et (4) tous les chemins reliant un nœud à une feuille (de ses descendants) contiennent le même nombre de nœuds noirs (= la *hauteur noire*). Ces propriétés interdisent un trop fort déséquilibre de l'arbre, sa hauteur reste en $\Theta(\log n)$.

Insertion d'une valeur : insérer le nœud avec la couleur rouge à la position qu'il aurait dans un arbre binaire de recherche classique, puis, si le père est rouge, considérer les trois cas suivants dans l'ordre.

Cas 1 : Le père et l'oncle du nœud considéré sont tous les deux rouges. **Répéter** cette transformation à partir du grand-père si l'arrière grand-père est aussi rouge.

Cas 2 : Si le père est rouge, l'oncle noir, et que le nœud courant n'est pas dans l'axe père–grand-père, une rotation permet d'aligner fils, père, et grand-père.

Cas 3 : Si le père est rouge, l'oncle noir, et que le nœud courant est dans l'axe père–grand-père, un rotation et une inversion de couleurs rétablissent les propriétés des ARN.



(Dans tous ces cas, les lettres grecques représentent des sous-arbres avec la même hauteur noire.)