Correction

Examen d'algorithmique

EPITA ING1 2011 S1; A. DURET-LUTZ

Durée : 1 heure 30

Janvier 2009

Consignes

- Cet examen se déroule **sans document** et **sans calculatrice**.
- Répondez sur le sujet dans les cadres prévus à cet effet.
- Il y a 5 pages d'énoncé, et 1 page d'annexe dont vous ne devriez pas avoir besoin.
  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 22.

1 Dénombrement (5 points)

On considère des entiers positifs codés avec 8 bits de façon classique. Pour cet exercice on représentera même les 0 inutiles en tête des nombres. Par exemple vingt-trois se représente 00010111.

- Un entier est dit *équitable* si la moitié de ses bits sont des 1 et l'autre des 0. Par exemple 00010111 et 11011000 sont équitables, alors que 10001001 ne l'est pas.
- Un entier équitable sera dit *convenable* si n'importe lequel de ses suffixes possède au moins autant de 1 que de 0. Par exemple 00010111 est convenable (ses suffixes sont 1, 11, 111, 0111, 10111, 010111, 0010111 et lui-même), mais 00111001 n'est pas convenable (le suffixe 001 possède strictement plus de 0 que de 1).

Si nous remplaçons les 0 et 1 respectivement par des parenthèses ouvrantes et fermantes, les nombres convenables de 8 bits permettent de représenter l'ensemble des chaînes constituées de quatre paires de parenthèses correctement assemblées (i.e., les parenthèses ne sont jamais fermées avant d'avoir été ouvertes). Par exemple 00010111 correspond à "`((()())))`".

1. **(1 point cadeau)** Combien existe-t-il d'entiers codés avec 8 bits ?

Réponse :

Il y a $2^8 = 256$ entiers sur 8 bits.
On s'imagine que tout informaticien devrait connaître cette valeur. La correction des copies révèle de surprenantes perles : 128, 255, 512, $2^{8-1}+1 = 129$, $2^{8+1} = 256$, ou encore $2^9 - 1 = 255$. Notez que la même question se trouvait dans le partiel de THL de Janvier 2008 et que les réponses y étaient encore plus farfelues.

2. **(2 points)** Combien existe-t-il d'entiers *équitables* sur 8 bits ?

Réponse :

La position des bits à 1 peut être déterminée en choisissant 4 indices parmi les 8 disponibles.
Il y a donc $C_8^4 = \frac{8\times7\times6\times5}{4\times3\times2\times1} = 2\times7\times5 = 70$ possibilités.

3. **(2 points)** Combien existe-t-il d'entiers *convenables* sur 8 bits ?

Réponse :

Plusieurs façons d'en venir à bout :

À partir des entiers équitables Considérons les entiers équitables qui ne sont pas convenables. Dans ces entiers, trouvons le suffixe qui empêche un nombre d'être convenable. Ce suffixe commence forcément par 0. Par exemple dans 00111001, c'est dès 001 que les 0 sont en excès. Inversons toutes les chiffres qui ne sont pas dans ce suffixe. Par exemple 00111001 devient 11000001. Le mot ainsi construit contient obligatoirement trois occurrences du chiffre 1 et cinq occurrence du chiffre 0. Tous les mots de possédant trois 1 et cinq 0 peuvent être obtenus par ce moyen et de façon unique. Il y a C_8^3 tels nombres (équitables mais pas convenables), donc il y a exactement $C_8^4 - C_8^3 = 70 - 56 = 14$ nombres convenables.

À partir de la représentation sous forme de chaîne de parenthèses – Une chaîne de 8 parenthèses peut être :
- `(6)` deux parenthèses autour d'une chaîne de 6 parenthèses
- `(4)2` deux parenthèses autour d'une chaîne de 4 parenthèses puis une chaîne de 2 parenthèses
- `(2)4` deux parenthèses autour d'une chaîne de 2 parenthèses puis une chaîne de 4 parenthèses
- `()6` deux parenthèses puis une chaîne de 4 parentheses

– Une chaîne de 6 parenthèses peut-être :
- `(4)` deux parenthèses autour d'une chaîne de 4 parenthèses
- `(2)2` deux parenthèses autour d'une chaîne de 2 parenthèses puis une chaîne de 2 parenthèses
- `()4` deux parenthèses puis une chaîne de 4 parenthèses

– Une chaîne de 4 parenthèses peut-être :
- `(2)` deux parenthèses autour d'une chaîne de 2 parenthèses
- `()2` deux parenthèses puis une chaîne de 2 parenthèses

– Une chaîne de 2 parenthèses ne peut-être que `()`.

On en déduit, en notant $T(n)$ le nombre de possibilités de faire une chaîne de n parenthèses, que :

$$\begin{aligned} T(2) &= 1 \\ T(4) &= T(2) + T(2) = 2 \\ T(6) &= T(4) + T(2) \times T(2) + T(4) = 5 \\ T(8) &= T(6) + T(4) \times T(2) + T(2) \times T(4) + T(6) = 14 \end{aligned}$$

Il y a donc 14 entiers convenables sur 8 bits.

Dans le cas général (qui n'était pas demandé), il existe C_{2n}^n entiers *équitables* sur $2n$ bits, et $C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$ entiers *convenables*. Vous devriez reconnaître cette valeur comme étant le n^{e} nombre de Catalan, qui est apparu en cours lorsque nous avons compté le nombre de parenthésages dans les chaînes de multiplication de matrices. En fait on peut faire une correspondance entre la liste des nombres *convenables* sur $2n$ bits et l'ensembles des parenthésages possibles d'une chaîne de multiplication de $n+1$ matrices. Par exemple le nombre 00011011 peut être vu comme la chaîne de parenthèses "`((())())`", soit la chaîne de multiplications $(((A_1A_2)A_3)(A_4A_5))$.

2 Plus longue sous-séquence croissante (8 points)

Étant donné une séquence de n entiers différents, on souhaite y trouver l'une des *plus longues sous-séquences croissantes*. Par exemple si la séquence fournie est $[10, 53, 85, 35, 51, 52, 74, 52, 07, 20, 71, 75, 13]$, alors la plus longue sous-séquence croissante est $[10, 35, 51, 52, 52, 71, 75]$. (Notez que la croissante n'est pas forcément stricte.) Si plusieurs sous-séquences possèdent la taille la plus longue, l'algorithme peut retourner n'importe laquelle.

Dans ce qui suit, on note S la séquence d'entrée complète (de n éléments numérotés de 1 à n), $S[i]$ son i^{e} élément et $S[1..i]$ son i^{e} préfixe (c'est-à-dire la séquence constituée seulement des i premiers éléments).

Premier algorithme (2 points)

Une première idée est de se ramener à l'algorithme de recherche de la plus longue sous-séquence *commune* à deux chaînes (ou séquences) que nous avons vu en cours. Pour deux chaînes u et v de tailles $|u|$ et $|v|$, l'algorithme du cours produit l'une des plus longues sous-séquences communes avec la complexité $\Theta(|u| \cdot |v|)$.

La recherche de la plus longue sous-séquence croissante peut alors s'effectuer ainsi :

PlusLongueSousSéquenceCroissante(S) :
    retourner PlusLongueSousSéquenceCommune(S, TriCroissant(S))

(2 points) Donnez le nom d'un algorithme de tri qui pourrait être utilisé ici et indiquez la complexité de cet algorithme de tri. Donnez ensuite la complexité de PlusLongueSousSéquenceCroissante pour l'algorithme de tri que vous avez choisi. (Vous donnerez ces complexités en fonction de $n = |S|$.)

Réponse :

Si l'on utilise un *tri par tas*, qui possède une complexité de $\Theta(n \log n)$, la complexité globale de PlusLongueSousSéquenceCroissante est $\Theta(n \log n) + \Theta(n^2) = \Theta(n^2)$.
En fait, la complexité de PlusLongueSousSéquenceCroissante est $\Theta(n^2)$ avec n'importe lequel des algorithmes de tri comparatif vus en cours, car le pire des algorithme que nous avons vus est en $\Theta(n^2)$ (c'est le tri par sélection).

Deuxième algorithme (6 points)

On souhaite maintenant développer un algorithme de programmation dynamique qui va trouver la plus longue sous-séquence croissante sans avoir besoin ni d'effectuer un tri, ni d'appeler PlusLongueSousSéquenceCommune.

Pour $k \leq n$, notons $L[k]$ la longueur de la plus longue sous-séquence croissante de $S[1..k]$ qui se termine par $S[k]$. On a bien sûr $L[1] = 1$. Selon l'ordre de $S[1]$ et $S[2]$, la valeur de $L[2]$ sera soit $L[1] + 1$, soit 1.

1. **(2 points)** Donnez une définition récursive de $L[k]$ (en fonction des termes précédents de L, et des éléments de S).

Réponse :

Supposons que l'on connaisse $L[1], L[2], \ldots, L[k-1]$ et que l'on veuille calculer $L[k]$, c'est-à-dire la plus longue sous-séquence croissante de $S[1..k]$. Notons $Q_k = \{i \in [\![1, k[\![\mid S[i] \leq S[k]\}$ l'ensemble des positions de S contenant une valeur inférieure ou égale à $S[k]$. Lorsque Q_k n'est pas vide il indique toutes les positions i pour lesquelles la plus longue sous-séquence croissante de $S[1..i]$ qui se termine en i peut être complétée par $S[k]$. On a alors $L[k] = 1 + \max\{L[i] \mid i \in Q_k\} = \max\{1 + L[i] \mid i \in Q_k\}$. En revanche, si Q_k est vide, alors $S[k]$ ne peut compléter aucune sous-séquence croissante et $L[k] = 1$. Ces remarques nous donnent :

$$\forall k \geq 1, \quad L[k] = \max\left(\{1\} \cup \{L[i] + 1 \mid i \in [\![1, k[\![\text{ et } S[i] \leq S[k]\}\right)$$

2. **(1 point)** Une fois que les $L[1], L[2], \ldots, L[n]$ ont été calculés, comment peut-on trouver la taille de la plus longue sous-séquence croissante de S ? (Cette plus longue sous-séquence ne se termine pas forcément par $S[n]$.)

Réponse :

Cette taille est bien sûr le maximum de tous les $L[i]$ calculés :

$$T = \max\{L[i] \mid i \in [\![1, n]\!]\}$$

Une autre façon de calculer T serait d'ajouter à S un nouvel élément qui majore tous les autres. Par exemple posons $S[n+1] = \infty$. De cette façon on a $T = L[n+1] - 1$.

3. **(1 point)** Quelle est la complexité de calculer la taille de la plus longue sous-séquence croissante d'une séquence de n éléments, avec un algorithme de programmation dynamique basé sur les définitions précédentes ?

Réponse :

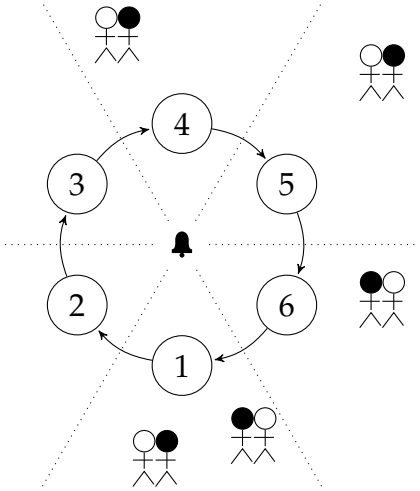
Il faut calculer $L[1..n]$, et chaque calcul de $L[k]$ demande de parcourir $L[1..k-1]$ pour trouver la valeur maximale. Cette première étape prend donc $\Theta(n^2)$ opérations. Il faut ensuite rechercher le maximum de $L[1..n]$ ce qui prend $\Theta(n)$ opérations. La complexité totale de l'algorithme est $\Theta(n^2)$.

4. **(2 points)** Comment faudrait-il adapter cet algorithme pour retourner, en plus de la longueur de la plus longue sous-séquence croissante, la sous-séquence croissante elle-même ? (On ne vous demande pas d'écrire l'algorithme, mais seulement d'expliquer ce qu'il doit faire en plus.)

Réponse :

Pour chaque $L[k]$ l'algorithme doit garder trace du i qui a été retenu pour le calcul du max. (On peut convenir que $i = 0$ si ce que nous avions appelé Q_k est vide.) La valeur de ce i doit être sauvegardé dans un tableau, disons $P[k]$, chaque fois qu'un nouveau $L[k]$ est complété. À la fin de l'algorithme, on peut remonter les $P[k]$ pour reconstruire la plus longue sous-séquence croissante : si l'on a posé $S[n+1] = \infty$ comme dans la question 2, alors $P[n+1]$ indique le dernier élément de la séquence, $P[P[n+1]]$ l'avant-dernier élément de la séquence, etc.

3 Hachage de chaises (5 points)



On considère une pièce plongée dans le noir dans laquelle on a disposé m chaises en cercle autour d'une clochette. Dans la pièce, hors de ce cercle de chaises, se trouvent trouvent n couples de personnes immobiles. Il y a toujours plus de chaises que de couples ($n < m$). La figure ci-contre illustre le cas avec $m = 6$ et $n = 5$. Lorsque la clochette sonne, les hommes de chaque couple se dirigent vers la chaise la plus proche (il fait noir, mais il suffit de se diriger vers la clochette qui tinte). L'homme s'assied si la chaise est libre, sinon il doit essayer les chaises suivantes dans l'ordre des aiguilles d'une montre jusqu'à en trouver une libre pour s'y asseoir . Quand tous les hommes de notre exemple seront assis ils occuperont les chaises 1, 2, 4, 5 et 6.

La fonction qui à un couple associe la chaise la plus proche peut être vue comme une fonction de hachage. La présence d'une personne sur cette chaise est une collision. Si une femme veut retrouver son mari, elle doit se diriger vers la chaise la plus proche puis « tâter » les hommes assis un par un et dans le sens des aiguilles d'une montre, jusqu'à retrouver le bon.

1. **(2 points)** De quel type de hachage ce cercle de chaises fait-il l'analogie ? (1 point pour le nom du hachage et 1 point pour le type de sondage.)

Réponse :

Il s'agit d'*adressage ouvert* avec *sondage linéaire*.
Pour le type de hachage j'ai accepté *hachage fermé* comme synonyme d'*adressage ouvert*. J'ai aussi donné le point pour *hachage coalescent* même si c'est un peu différent (il y a une réserve).
Pour le sondage j'ai donné le point dès que j'ai lu le mot *linéaire* ou une formule qui donnait la façon de boucler avec un modulo.

2. **(1 point)** Quel est l'inconvénient principal de ce type de hachage ?

Réponse :

Plus une séquence d'emplacements occupés est longue, plus elle risque d'être allongée, augmentant les temps de recherche. C'est l'**effet de grappe**.
J'ai aussi donné le point à ceux qui m'ont expliqué que la suppression était plus compliquée.
Par contre tous ceux qui m'ont dit qu'*il y a des collision* n'ont pas eu de points car c'est un problème commun à tout type de hachage.

3. **(2 points)** On se restreint maintenant au cas où $m > 2$ et $n = 2$. C'est-à-dire qu'il n'y a que deux couples disposés aléatoirement dans la pièce. On supposera la loi de distribution des positions autour du cercle de chaises uniforme, autrement dit la probabilité d'être plus proche d'une chaise donnée est $1/m$.

   Une fois que les hommes sont assis, quelle est l'espérence du nombre d'hommes qu'une femme doit tâter avant de retrouver le sien ?

Réponse :

Si les couples sont associés à des chaises différentes, il n'y a qu'une comparaison à faire : ce cas a $m-1$ chances sur m de se produire.
Sinon les deux couples sont proches de la même chaise, et cette situation se produit avec une chance sur m. Dans ce cas il y a une chance sur deux que la personne recherchée se soit assise en premier, auquel cas il n'y a qu'un comparaison à faire. Il y a une chance sur deux que la personne recherchée se soit assise en second, et il y a alors 2 comparaisons à faire.
Si on note T le nombre d'hommes à tâter on a

$$E(T) = 1\Pr\{T=1\} + 2Pr\{T=2\} = 1 \cdot \left(\frac{m-1}{m} + \frac{1}{2m}\right) + 2 \cdot \frac{1}{2m} = \frac{2m+1}{2m}$$

4 Tas spécial (4 points)

Normalement, un tas de n éléments est représenté par un tableau de n cases. Le père de l'élément situé à l'indice i se trouve à l'indice $Parent(i) = \lfloor i/2 \rfloor$ et peut être accédé en temps constant.

Pour mémoire, voici l'algorithme d'insertion d'une valeur v dans un tas représenté par les n premiers éléments d'un tableau A.

```
HEAPINSERT(A, n, v)
1  i ← n + 1
2  A[i] ← v
3  while i > 1 and A[Parent(i)] < A[i] do
4      A[Parent(i)] ↔ A[i]
5      i ← Parent(i)
```

1. **(1 point cadeau)** Quelle est la complexité de cet algorithme lorsque le tas possède n éléments.

Réponse :

$O(\log n)$, c'est du cours.
(D'autre part cette complexité était indiquée dans l'annexe !)

2. **(2 points)** Imaginez maintenant que l'on remplace le tableau A par une liste doublement chaînée. L'accès au père de l'élément i se fait alors en $\Theta(i/2)$ opérations parce qu'il faut remonter la liste de $i/2$ positions.
   Quelle est la complexité de l'algorithme d'insertion lorsque A est une liste doublement chaînée ? **Justifiez votre réponse.**

Réponse :

Au pire l'insertion va nous forcer à partir de $A[n]$ et à remonter jusqu'à la racine. Depuis $A[n]$ on trouve $A[Parent(n)]$ en $\Theta(n/2)$ opérations. À partir de là on trouve ensuite $A[Parent(Parent(n))]$ en $\Theta(n/4)$ opérations. Au total on aura fait $\frac{n}{2^1} + \frac{n}{2^2} + \cdots + \frac{n}{2^{\lfloor \log n \rfloor}} = \Theta(n)$ opérations, c'est-à-dire remonté toute la liste.
La complexité de l'algorithme est donc $O(n)$.

3. **(1 point)** On considère le tas représenté par le tableau suivant :

13	9	12	3	6	8	5	2

   Donnez l'état du tas après les *suppressions* successives de ses trois plus grandes valeurs.

8	6	2	3	5

FIN

Notations asymptotiques

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

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

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

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

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	$\Theta(n!)$	
	$\Theta(n^n)$	

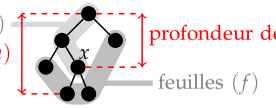
Théorème général

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 :

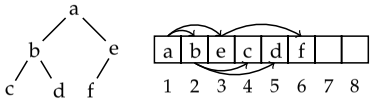
$n \leq 2^{h+1} - 1$ $\qquad$ $h \geq \lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$ si $n > 0$

$f \leq 2^h$ $\qquad$ $h \geq \lceil \log_2 f \rceil$ 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. $\mathrm{E}[X] = \sum_x \Pr\{X = x\}$

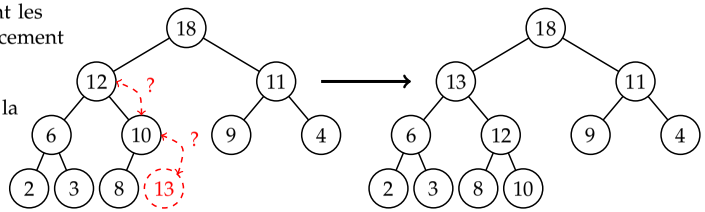
Variance : $\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - \mathrm{E}^2[X]$

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 $\mathrm{Var}[X_i] = np(1-p)$.

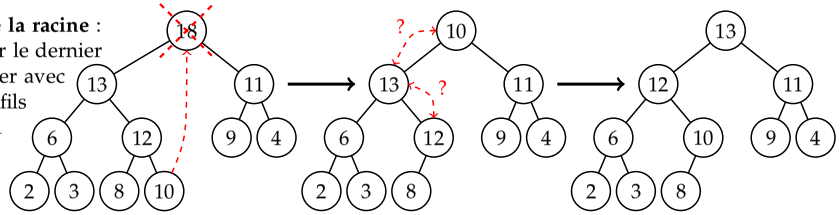
Tas

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

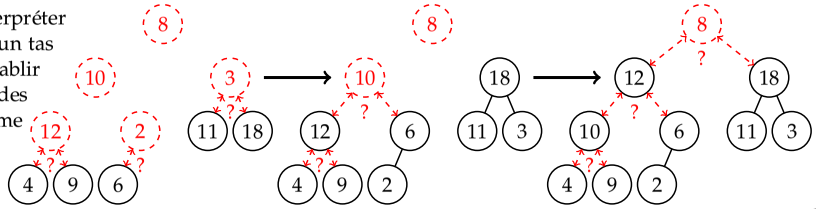
Dans les opérations qui suivent les arbres parfaits sont plus efficacement représentés par des tableaux.



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

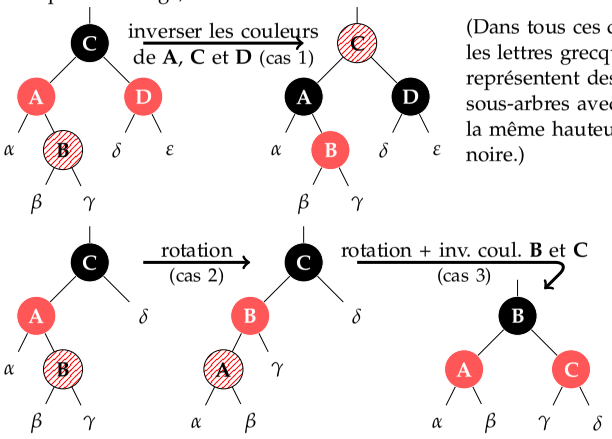
Les ARN sont des arbres binaires de recherche dans lesquels : (1) un nœud est rouge ou noir, (2) racine et feuilles (NIL) sont noires, (3) les fils d'un nœud rouge sont noirs, et (4) tous les chemins reliant un nœud à une feuille (de ses descendants) contiennent le même nombre de nœuds noirs (= la *hauteur 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.)