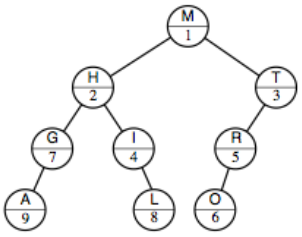
École normale supérieure **2010-2011**
Département d'informatique

Algorithmique et Programmation
TD n° 4 : Arbres
Avec Solutions

Arbre Binaire de Recherche Randomisé

Exercice 1. Un *treap* est un arbre binaire dans lequel chaque noeud a une *clé* et une *priorité*, où la suite des clés est ordonnée par ordre infixe et la priorité d'un noeud est plus petite que celle de ses fils. En d'autre terme un treap est simultanément un arbre binaire de recherche pour les clés et un tas (min) pour les priorités.



A treap. The top half of each node shows its search key and the bottom half shows its priority.

1. Montrer que la structure de l'arbre du treap est complètement déterminée par les clés et les priorités.
2. Montrer qu'un treap est l'arbre binaire de recherche qui résulte des insertions des clés dans l'ordre des priorités croissante.
3. Montrer comment réaliser les opérations et estimer leur coût en fonction de la profondeur d'un noeud (distance entre la racine et le noeud) et en fonction de n (le nombre total de noeuds) : rechercher, insérer/enlever (insérer $(S, 10)$ par exemple), un élément et séparer un treap T en deux treaps $T_<$ et $T_>$ tels que $T_<$ contient toutes les clés plus petite que la clé π et $T_>$ toutes les clés supérieures et fusionner deux treaps.

   Un *treap randomisé* est un treap dans lequel les priorités sont des variables aléatoires uniformément distribuée et indépendantes continues (pour éviter des priorités égales). On va montrer que la profondeur de tout noeud est $O(\log n)$. Soit x_k le noeud qui a la k-ième plus petite clé. On définit la variable indicatrice
   $$A_i^k = [x_i \text{ est un ancêtre propre de } x_k].$$
4. Exprimer la profondeur de x_k en fonction des variables indicatrices et estimer son espérance.

   On va maintenant estimer la probabilité qu'un noeud soit un ancêtre propre d'un autre. Soit $X(i,k)$ représente le sous-ensemble des noeuds $\{x_i, x_{i+1}, \ldots, x_k\}$ ou $\{x_k, x_{k+1}, \ldots, x_i\}$ selon que $i < k$ ou $k < i$.
5. Montrer que pour tout $i \neq k$, x_i est un ancêtre propre de x_k si et seulement si x_i a la plus petite priorité parmi tous les noeuds de $X(i,k)$.
6. Calculer la probabilité qu'un noeud soit un ancêtre propre d'un autre et en déduire la hauteur moyenne d'un noeud et donc le coût des opérations.
7. Pouvez-vous en déduire un nouvel algorithme de tri et montrer en quoi il ressemble à quicksort ?

Solutions : Nous allons prouver la question 1 par récurrence. Comme il s'agit d'un tas, le noeud v à la racine est celui qui a la priorité la plus petite. Comme il s'agit aussi d'un ABR, les noeuds u tels que $cle(u) < cle(v)$ sont dans le sous-arbre gauche et ceux qui ont ont une clé plus grandes que v sont dans le sous-arbre droit. Ensuite, comme la définition d'un ABR et d'un tas est récursive, les sous-arbres droit et gauche sont des ABR et des tas, et nous plaçons de façon unique chaque noeud.

La recherche dans un treap est la même que dans un ABR. Le temps est proportionnel à la recherche de ce noeud. Le cas de la recherche si le noeud ne se trouve pas dans le treap est le coût de la recherche de son successeur ou prédécesseur immédiat.

Pour insérer un noeud z, on comme par faire une insertion classique dans un ABR. Mais comme les priorités ne forment pas nécessairement un tas, on réordonne l'ABR en utilisant des rotations qui satisfont toujours la propriété d'ABR en repectant maintenant la propriété de tas.

Pour supprimer un noeud, on fait des rotations pour descendre le noeud sur une feuille ou de façon à ce qu'il est qu'un fils et ensuite, la suppression est simple.

Pour séparer un treap par rapport à une clé π, il suffit de faire remonter ce noeud avec des rotations à la racine du treap en lui mettant une priorité très faible. Puis, les sous-arbres gauche et droit forment des treap.

Pour joindre deux treaps tels que toutes les clés du premier sont inférieures à celle du second, il faut rajouter un noeud fantôme avec une priorité la plus faible possible à la racine des deux treaps à joindre et éliminer ensuite le noeud fantôme avec une opération de suppression.

La profondeur d'un noeud d'un treap est $\Theta(n)$ dans le pire cas et donc dans le pire cas, toutes les opérations ont un coût en $\Theta(n)$.

Un treap randomisé est un treap dans lequel les priorités sont choisies aléatoirement. Nous allons montrer que la profondeur est en moyenne en $O(\log n)$.

$$A_i^k = [x_i \text{ est un ancêtre propre de } x_k]$$

est une variable aléatoire indicatrice, c'est-à-dire valant 1 si x_i est un ancêtre de x_k et 0 sinon. (cf. le rappel de probabilité à la fin de ce corrigé.)

$$profondeur(x_k) = \sum_{i=1}^{n} A_i^k.$$

$$E[profondeur(x_k)] = \sum_{i=1}^{n} \Pr[A_i^k = 1].$$

Il suffit donc de calculer $\Pr[A_i^k = 1]$. Nous allons montrer que

$$\Pr[A_i^k = 1] = \frac{[i \neq k]}{|k-i|+1} = \begin{cases} \frac{1}{k-i+1} & \text{si} \quad i < k \\ 0 & \text{si} \quad i = k \\ \frac{1}{i-k+1} & \text{si} \quad i > k \end{cases}$$

Nous allons montrer que pour tout $i \neq k$, x_i est n ancêtre propre de x_k si et seulement si x_i à la plus petite priorité par rapport à tous noeuds dont la clé est comprise entre x_i et x_k, c'est-à-dire dans l'ensemble $X(i,k)$. Pour ce faire, nous montrons que le lemme est vrai quand x_i est à la racine, quand x_k est à la

racine, quand x_j est à la racine et x_i et x_k dans deux sous-arbres différents et quand x_j est à la racine et x_i et x_k dans le même sous-arbre.

Enfin,

$$\begin{aligned} E[profondeur(x_k)] = \sum_{i=1}^{n} \Pr[A_i^k = 1] &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\ &= \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k+1} \frac{1}{j} \\ &= H_k - 1 + H_{n-k+1} - 1 \\ &< \ln k + \ln(n-k+1) - 2 \\ &< 2\ln n - 2. \end{aligned}$$

En conclusion, toutes les opérations s'effectuent en temps $O(\log n)$ en moyenne. Comme un treap est exactement l'ABR qui résulte de l'insertion des clés en suivant l'ordre croissant des priorités, un treap randomisé est le résultat de l'insertion des clés dans un ordre aléatoire. L'analyse montre aussi que la profondeur moyenne d'un noeud dans un ABR construit en utilisant des insertions aléatoires est aussi $O(\log n)$.

Enfin, si on range dans un ABR des valeurs en utilisant un ordre d'insertion aléatoire, et qu'on lit l'arbre par ordre infixe, on obtient la liste triée. On remarque que l'insertion à la racine consiste à partitionner les entiers plus petit que la racine à gauche et plus grand à droite. C'est exactement ce qu'effectue Quicksort durant l'opération de partitionnement.

Splay Trees

Exercice 2. Le but de cet exercice est d'étudier un type d'arbre binaire de recherche qui a de bonne propriété amortie en autorisant des modifications de l'arbre aussi pendant la recherche d'un élément.

On rappelle que dans une analyse amortie avec la méthode du potentiel, on définit une fonction potentiel Φ pour la structure de donnée qui initialement vaut $\Phi_0 = 0$ et est toujours strictement positive $\Phi_i > 0$. Le coût amorti d'une opération est son coût plus le changement de potentiel : $a_i = c_i + \Phi_i - \Phi_{i-1}$. Il est facile de voir que si la fonction potentiel est *valide*, *i.e.* $\Phi_i - \Phi_0 \geq 0$ pour tout i, alors le coût total de toute suite d'opération est inférieur à son coût total amorti : $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$.

On va utiliser des rotations simples et les deux rotations doubles suivantes.

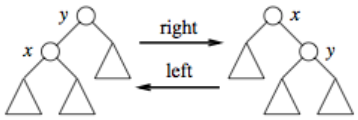


Figure 1. A right rotation at x and a left rotation at y are inverses.

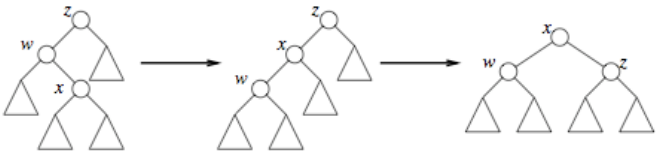


Figure 2. A zig-zag at x. The symmetric case is not shown.

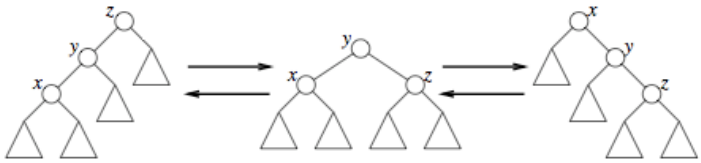


Figure 3. A right roller-coaster at x and a left roller-coaster at z.

Une opération *splay* déplace un noeud arbitraire dans l'arbre jusqu'à la racine par une suite de double rotation qui se termine éventuellement par une simple rotation à la fin. Un *splay tree* est un arbre binaire de recherche qui est conservé à peu près équilibré par les opérations splay. Après chaque accès à un noeud on le bouge vers la racine : en recherchant un noeud, s'il existe, on le splay sinon son prédécesseur ou successeur si la clé n'est pas présente, après une insertion, on splay le noeud, pour supprimer un noeud, on le splay, le supprime et remplace la racine par son prédécesseur immédiat.

Pour estimer la complexité de ces opérations, il est clair que la descente dans l'arbre est moins coûteuse que l'opération splay et donc on a seulement d'avoir une bonne borne sur cette opération. On définit le *rang* d'un noeud v par $rang(v)$ par $\lfloor \log taille(v) \rfloor$ et le potentiel $\Phi = \sum_v rang(v)$. On notera $r(v)$ (resp. $r'(v)$) le rang avant (resp. après) une rotation simple ou double.

1. Montrer que le coût amorti d'une simple rotation à partir de tout noeud v est au plus $1 + 3r'(v) - 3r(v)$ et le coût amortie d'une double rotation au noeud v est au plus $3r'(v) - 3r(v)$.
2. En déduire le coût amorti d'une opération splay.
3. Que peut-on dire du coût amorti d'accès à x si on y accède $f(x)$ fois avec $F = \sum_x f(x)$?

Solution : Chaque recherche, insertion, ou suppression, consiste en un nombre constant d'opérations de la forme descente jusqu'à un noeud et le remonter à l'aide d'opération splay jusqu'à la racine. Comme la descente est moins chère que la remontée, nous avons besoin d'avoir une bonne borne amortie pour les opérations de splay.

Nous définissons le *rang* d'un noeud v comme le $\lfloor \log size(v) \rfloor$ et le potentiel

$$\Phi = \sum_v rang(v) = \sum_x \lfloor \log size(v) \rfloor.$$

Il n'est pas difficile de voir que pour un arbre parfaitement équilibré, le potentiel est $\Theta(n)$ et $\Theta(n \log n)$ pour un arbre parfaitement mal équilibré.

En ajoutant les coûts amortis de toutes les rotations, nous trouvons que le coût amorti total des opérations splay à un noeud v est au plus $1+3rang'(v)-rang(v)$, où $rang'(v)$ est le rang de v après l'opération splay. Les rangs intermédiaires disparaissent à cause de la somme télescopique. Après toutes les opérations, v devient la racine de l'arbre. Donc, $rang'(v) = \lfloor \log n \rfloor$, ce qui implique que le coût amorti est au plus $3 \log n - 1 = O(\log n)$. Nous en déduisons que toute insertion, suppression, ou recherche prend un temps $O(\log n)$ en moyenne.

Les splay trees sont optimaux en plusieurs sens. Nous allons généraliser le résultat précédent pour des poids réels positifs $w(x)$ sur chaque noeud x. Dans le résultat précédent, pour ces poids :

$$s(v) = w(v) + s(sag(v)) + s(sad(v)),$$

et $r(v) = \lfloor \log s(v) \rfloor$. Dans le cas des tailles, nous avons $w(v) = 1$.

Cas 1 : simple rotation :

$$\begin{array}{rcll} 1 + \Phi' - \Phi & = & 1 + r'(x) + r'(y) - r(x) - r(y) & [\text{seuls } x \text{ et } y \text{ change de rang}] \\ & \leq & 1 + r'(x) - r(x) & [r'(y) \leq r(y)] \\ & \leq & 1 + 3r'(x) - 3r(x) & [r'(x) \geq r(x)] \end{array}$$

Cas 2 : zig-zag :

$$\begin{array}{rcll} 2 + \Phi' - \Phi & = & 2 + r'(w) + r'(x) + r'(z) - r(w) - r(x) - r(z) & [\text{seuls } w, x, z \text{ changent de rang}] \\ & \leq & 2 + r'(w) + r'(x) + r'(z) - 2r(x) & [r(x) \leq r(w) \text{ et } r'(x) = r(z)] \\ & = & 2 + (r'(w) - r'(x)) + (r'(z) - r'(x)) + 2(r'(x) - r(x)) & \\ & = & 2 + \log \frac{s'(w)}{s'(x)} + \log \frac{s'(z)}{s'(x)} + 2(r'(x) - r(x)) & \\ & \leq & 2 + 2 \log \frac{s'(x)/2}{s'(x)} + 2(r'(x) - r(x)) & [s'(w) + s'(z) \leq s'(x), \log \text{ est concave}] \\ & = & 2(r'(x) - r(x)) & \\ & \leq & 3(r'(x) - r(x)) & [r'(x) \geq r(x)] \end{array}$$

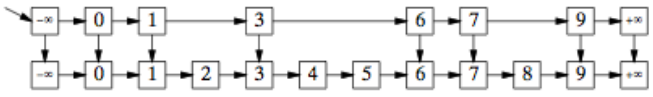
Cas 3 : roller-coaster :

$$\begin{array}{rcll} 2 + \Phi' - \Phi & = & 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) & [\text{seuls } x, y, z \text{ changent de rang}] \\ & \leq & 2 + r'(x) + r'(z) - 2r(x) & [r'(y) \leq r(z) \text{ et } r(x) \leq r(y)] \\ & = & 2 + (r(x) - r'(x)) + (r'(z) - r'(x)) + 3(r'(x) - r(x)) & \\ & = & 2 + \log \frac{s(x)}{s'(x)} + \log \frac{s'(z)}{s'(x)} + 3(r'(x) - r(x)) & \\ & \leq & 2 + 2 \log \frac{s'(x)/2}{s'(x)} + 3(r'(x) - r(x)) & [s(x) + s'(z) \leq s'(x), \log \text{ est concave}] \\ & = & 3(r'(x) - r(x)) & \end{array}$$

En prenant $w(v) = f(v)/F$, nous en déduisons que le coût pour v est $O(logF - logf(v))$, ce qui est le coût pour un ABR optimal $\log(f(x)/F)$.

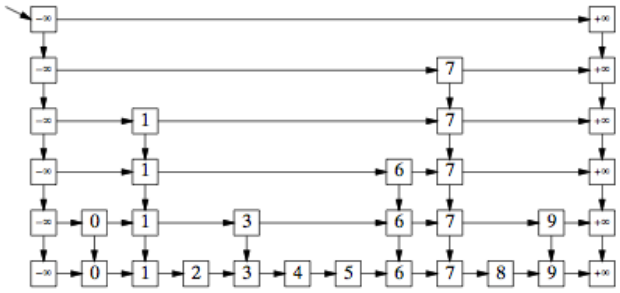
Listes à saut

Exercice 3. Une liste à saut est simplement une liste chaînée triée avec quelques raccourcis aléatoires. La recherche dans une liste simplement chaînée de taille n demande $O(n)$ dans le pire cas. Pour accélérer la recherche, on pourrait avoir une deuxième liste qui contient à peu près la moitié des éléments de la première, en dupliquant chacun avec probabilité $1/2$. On chaîne ensemble tous les doublons en une deuxième liste et on ajoute un pointeur de chaque doublon vers l'original, avec des sentinelles à chaque bouts.



A linked list with some randomly-chosen shortcuts.

On peut ensuite utiliser récursivement la même idée, ce qui nous donne :



A skip list is a linked list with recursive random shortcuts.

1. Estimer le coût moyen d'une recherche dans le cas avec deux listes.
2. Écrire l'algorithme de recherche d'un élément.
   Supposons que les clés soient les éléments 1 à n. Soit $L(x)$ le nombre de niveaux qui contiennent la clé x, sans compter la liste en bas.
3. Calculer en moyenne le nombre de niveaux $E_x[L(x)]$.
   Pour estimer le pire cas moyen, on a besoin d'une borne sur $L = \max_x L(x)$. On ne peut pas faire comme dans le cas de la somme, on va déterminer un résultat plus fort :
4. Montrer que la hauteur est $O(\log n)$ avec forte probabilité, *i.e.* supérieure à $1 - 1/n^c$ pour une constante $c > 1$. La constante cachée dans le O dépend de c.
5. En déduire une borne sur le nombre moyen de niveaux de $L(x)$ pour un x fixé.
6. Estimer le coût de la recherche d'un élément.

Solution :

On remarque que s'il y a deux niveaux, nous allons faire au plus $n/2$ comparaisons pour le niveau haut et $1 + \sum_{k\geq 0} 2^{-k}$ en moyenne pour le niveau bas. En effet, la probabilité qu'un noeud soit suivi de k noeuds sans duplication au niveau haut est 2^{-k}.

Intuitivement, comme chaque niveau d'une skip list a moitié moins d'éléments que le niveau précédent, le nombre total de niveau est environ $O(\log n)$. De manière similaire, chaque fois que nous ajoutons un nouveau niveau, nous divisons par 2, plus une petite constante, le temps d'exécution, et donc si on a $O(\log n)$ niveaux, le temps sera également $O(\log n)$.

Algorithm 1 RechercheSkipList

Require: Clé x dans la liste L
Ensure: Le noeud contenant x

```
1: v ← L
2: while v ≠ NULL and cle(v) ≠ x do
3:     if cle(droit(v)) > x then
4:         v ← down(v)
5:     else
6:         v ← droit(v)
7:     end if
8: end while
9: return v
```

Le nombre de niveaux moyen est $E_x[L(x)]$ et vérifie avec probabilité $1/2$, $L(x) = 0$ et avec probabilité $1/2$, $L(x)$ est augmenté de 1. Donc,

$$E_x[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot (1 + E_x[L(x)]).$$

En résolvant cette équation, nous obtenons $E_x[L(x)] = 1$.

Pour analyser dans le pire cas la recherche d'un élément, nous avons besoin d'une borne sur le nombre de niveau $L = \max_x L(x)$. Malheureusement, nous ne pouvons pas calculer la moyenne d'un max comme la moyenne d'une somme. Nous allons obtenir un résultat plus fort en montrant que la profondeur est $O(\log n)$ avec forte probabilité.

Une clé x apparaît au niveau ℓ si on a eu ℓ piles lors du tirage de probabilité pour ce x. Par conséquent, $\Pr[L(x) \geq \ell] = 2^{-\ell}$. La skip list a au moins ℓ niveaux, si et seulement si $L(x) \geq \ell$ pour au moins une des n clés.

$$\Pr[L \geq \ell] = \Pr[(L(1) \geq \ell) \vee (L(2) \geq \ell) \vee \ldots \vee (L(n) \geq \ell)].$$

En utilisant la borne de l'union, $\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$,

$$\Pr[L \geq \ell] \leq \sum_{x=1}^{n} \Pr[L(x) \geq \ell] n \cdot \Pr[L(x) \ \geq \ell] = \frac{n}{2^\ell}.$$

Quand $\ell \leq \log n$, cette borne est triviale. Cependant, pour toute constante $c > 1$, nous avons la borne

$$\Pr[L \geq c \log n] \leq \frac{1}{n^{c-1}}.$$

Nous pouvons en conclure qu'avec forte probabilité, la skip list a $O(\log n)$ niveaux.

Cette borne est utile pour borner la hauteur d'un noeud fixé. En effet,

$$E[L] = \sum_{\ell \geq 0} \ell \cdot \Pr[L = \ell] = \Pr[\ell \geq 1] \Pr[L \geq \ell].$$

Si $\ell < \ell'$, alors $\Pr[L(x) \geq \ell] > \Pr[L(x) \geq \ell']$ et donc,

$$\begin{array}{rcll} E[L(x)] & = & \sum_{\ell \geq 1} \Pr[L \geq \ell] & \\ & = & \sum_{\ell=1}^{\log n} \Pr[L \geq \ell] + \sum_{\ell \geq \log n + 1} \Pr[L \geq \ell] & \\ & \leq & \sum_{\ell=1}^{\log n} 1 + \sum_{\ell \geq \log n + 1} \frac{n}{2^\ell} & \\ & = & \log n + \sum_{i \geq 1} \frac{1}{2^i} & [i = \ell - \log n] \\ & = & \log n + 1 & \end{array}$$

Donc, en moyenne, la skip list a au plus 1 niveau par rapport à la version idéale où chaque niveau contient exactement la moitié des noeuds du niveau précédent.

1 Rappels de probabilité

Une variable aléatoire réelle est une application d'un univers Ω vers $\mathbb{R}$ muni d'une probabilité p. Cette application crée un nouvel univers $X(\Omega)$ de réels sur lequel on peut construire une probabilité issue de p. Cette probabilité s'appelle la loi de probabilité de X.

L'espérance d'une variable aléatoire X à valeurs dans un ensemble S discret est

$$E[X] = \sum_{s \in S} s \cdot \Pr[X = s].$$

L'espérance est linéaire, c'est-à-dire si $X = \sum_{i=1}^{n} a_i X_i$, $E[X] = \sum_{i=1}^{n} a_i E[X_i]$. La variance $V(X)$ d'une variable aléatoire X est l'espérance de la variable aléatoire $(X - E[X])^2$, c'est-à-dire $E[(X - E[X])^2] = E[X^2] - (E[X])^2$. L'écart-type est la valeur $\sigma_X = \sqrt{V(X)}$.

Theorem 1.1. ***(Inégalité de Markov) :*** *Soit X une variable aléatoire positive et $v > 0$, alors*

$$\Pr[X \geq v] \leq E[X]/v$$

En effet, nous avons

$$\begin{aligned} E[X] &= \sum_{x \geq 0} \Pr[X = x] \cdot x \\ &\geq \sum_{0 \leq x < v} \Pr[X = x] \cdot 0 + \sum_{x \geq v} \Pr[X = x] \cdot v \\ &= \Pr[X \geq v] \cdot v. \end{aligned}$$

Quand nous avons des informations sur la variance, nous pouvons utiliser l'inégalité suivante :

Theorem 1.2. ***(Inégalité de Chebyshev) :*** *Soit X une variable aléatoire et $\delta > 0$. Alors*

$$\Pr[|X - E(X)| \geq \delta] \leq \frac{V[X]}{\delta^2}$$

En effet, la variable aléatoire $Y = (X - E[X])^2$ est positive et d'après Markov,

$$\begin{aligned} \Pr[|X - E[X]| \geq \delta] &\leq \Pr[(X - E[X])^2 \geq \delta^2] \\ &\leq \frac{E[(X - E[X])^2]}{\delta^2} \\ &= \frac{V[X]}{\delta^2}. \end{aligned}$$

Theorem 1.3. ***(Bornes de Chernoff) :*** *Soit $X_1, X_2, \ldots, X_n$, n essais deux à deux indépendants Bernoulli tels que $P(X_i) = p$ et de même variance σ. Soit $X = \sum_{i=1}^{n} X_i$ et $\mu = E(X) = n \cdot p$. Alors les inégalités suivantes sont vérifiées :*

1. *Pour tout $\delta > 0$, $\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$*
2. *Pour tout $\delta > 0$, $\Pr[X < (1-\delta)\mu] < \left(\frac{e^\delta}{(1-\delta)^{1-\delta}}\right)^\mu < e^{-\mu\delta^2/2}$*
3. *Pour $0 < \delta < 1$, $\Pr[X > (1+\delta)\mu] < e^{-\mu\delta^2/3}$*

Appliquer Chebyshev à la variable aléatoire $X = \sum_{i=1}^{n} X_i$