

---

**TD 09 – Suspensions**

---


**Exercice 1.***RAZ de marée*

On analyse des opérations sur un compteur binaire sur  $k$  bits qui commence à zéro. Ce compteur est représenté par un tableau  $A[0..k-1]$  de bits, et un nombre  $x$  représenté par le compteur est tel que  $x = \sum_{i=0}^{k-1} A[i].2^i$ .

1. Rappeler le principe d'une opération INCRÉMENTER sur ce compteur, qui rajoute 1 modulo  $2^k$  à la valeur actuelle du compteur. Montrer que si l'on disposait également d'une opération DÉCRÉMENTER sur le compteur, alors une suite de  $n$  opérations pourrait coûter jusqu'à un temps en  $O(nk)$ .
2. On garde uniquement l'opération INCRÉMENTER, et on rajoute au compteur une opération RÀZ qui remet le compteur à zéro. Montrer comment implémenter ce compteur sous la forme d'un tableau de bits pour qu'une séquence quelconque de  $n$  opérations INCRÉMENTER et RÀZ prenne un temps  $O(n)$  sur un compteur initialement à zéro.  
*Conseil : Gérer un pointeur vers le 1 de poids fort.*

**Exercice 2.***T'amortis ?*

On considère une structure de donnée de tas min à  $n$  éléments, avec les deux opérations INSÉRER et EXTRAIRE-MIN fonctionnant en temps  $O(\log n)$  dans le pire cas.

-  Trouver une fonction potentiel  $\Phi$  tel que le coût amorti de INSÉRER soit  $O(\log n)$  et celui de EXTRAIRE-MIN de  $O(1)$ , et prouver sa correction.

**Exercice 3.**

Cet exercice est tiré de l'article [1].

Considérons le problème de réaliser une séquence d'accès sur un ensemble d'éléments muni d'une relation d'ordre total. L'accès à un élément prend en entrée une clé, et retourne soit l'élément concerné avec ses informations s'il existe, soit une information précisant que l'élément n'existe pas. Une façon de résoudre ce problème est d'utiliser des *arbres binaires de recherche* : pour tout nœud  $x$  le sous-arbre gauche contient tous les éléments plus petits que  $x$  et le sous-arbre droit ceux plus grands. Le temps d'accès à un élément est en  $O(p)$ , où  $p$  est la profondeur de l'élément. Supposons que l'on souhaite réaliser  $m$  accès successifs à des éléments de l'ensemble, afin de réduire le coût total d'accès il faut que les éléments les plus fréquemment touchés soient près de la racine.

**Principe** Les arbres déployés sont une forme d'arbres binaires de recherche auto-ajustables. Ils remontent près de la racine les éléments qui sont régulièrement accédés. Pour cela, l'heuristique *déployer* (*splaying*) est utilisée. Elle réalise des rotations entre un élément et son père (ou son père et son grand-père), afin de le faire remonter jusqu'à la racine. La rotation diffère en fonction de la forme de l'arbre :

- Cas 1 (*zig*, figure 1a) : si  $p(x)$  le père de  $x$ , est la racine, alors retourner l'arête joignant  $x$  à  $p(x)$ .
- Cas 2 (*zig-zig*, figure 1b) : si  $p(x)$  n'est pas la racine, et que  $x$  et  $p(x)$  sont tous les deux des fils gauches ou droits, alors retourner l'arête joignant  $p(x)$  avec le grand-père  $g(x)$ , puis retourner l'arête joignant  $x$  avec  $p(x)$
- Cas 3 (*zig-zag*, figure 1c) : si  $p(x)$  n'est pas la racine, et que  $x$  est un fils gauche et  $p(x)$  est un fils droit (ou inversement), alors retourner l'arête joignant  $x$  avec  $p(x)$ , puis retourner l'arête joignant  $x$  avec le nouveau  $p(x)$ .

Lorsque l'on déploie un élément  $x$ , on applique autant de fois que nécessaire *zig-zig*, *zig-zig* ou *zig-zag* afin de faire remonter  $x$  à la position de la racine.

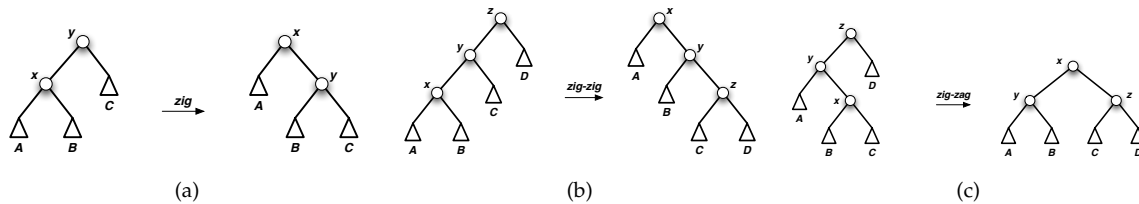


FIGURE 1 – Les différentes étapes pouvant intervenir durant le déploiement : 1a zig, 1b zig-zig et 1c zig-zag.

1. Appliquer *déployer* sur le nœud  $a$  de l'arbre figure 2.
2. Quelle est la complexité de l'exécution de l'heuristique *déployer* sur un nœud  $x$  ?

**Propriétés** On définit le potentiel d'un arbre déployé de la façon suivante. Chaque élément  $i$  possède un poids positif  $w(i)$ , sa valeur est arbitraire, mais fixée. On définit la taille  $s(x)$  d'un nœud  $x$  dans l'arbre comme étant la somme des poids de tous les éléments présents dans le sous-arbre dont la racine est  $x$ . On définit également le rang  $r(x)$  d'un nœud  $x$  comme étant  $\log(s(x))$ . Enfin, on définit le potentiel d'un arbre comme étant la somme des rangs de tous ses nœuds. Le temps d'exécution d'une opération *déployer* sera le nombre de rotations effectuées, et s'il n'y a pas de rotation alors le coût sera de 1.

3. Montrer que le coût amorti de l'opération *zig* peut être majoré par  $1 + 3(r'(x) - r(x))$ , et les opérations *zig-zig* et *zig-zag* par  $3(r'(x) - r(x))$  (avec  $r(x)$  le rang de  $x$  avant l'opération, et  $r'(x)$  le rang après). On pourra utiliser le fait que si on a  $x + y \leq 1$ , alors  $\log(x) + \log(y) \leq -2$ .
4. En déduire le **lemme d'accès** : le temps amorti pour l'exécution de *déployer* sur un nœud  $x$  d'un arbre enraciné en  $t$  est d'au plus  $3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$ .

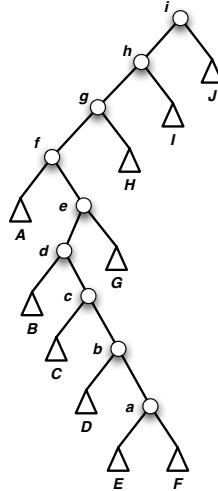


FIGURE 2 – Exemple pour *déployer*

Le poids des éléments étant un paramètre de l'analyse et non de l'algorithme, le lemme est valide pour n'importe quel poids positif. On va voir dans la suite qu'en choisissant intelligemment ces poids on peut obtenir des résultats intéressants sur les arbres déployés. On considère maintenant une suite de  $m$  accès, et on pose  $W = \sum_{i=1}^n w(i)$ . Pour les questions suivantes, on choisira les  $w(i)$  de façon à avoir  $W = O(1)$ .

5. Montrer le **théorème d'équilibre** : le temps total d'accès est  $O((m+n) \log n + m)$ .

Ce théorème affirme que sur une séquence suffisamment longue d'accès un arbre déployé est aussi efficace que n'importe quelle forme d'arbre uniformément équilibré.

Pour chaque élément  $i$ , on définit  $q(i)$  comme étant la fréquence d'accès de  $i$ , c'est-à-dire le nombre de fois que  $i$  est accédé.

6. Montrer le **théorème d'optimalité statique** : si tous les objets sont accédés au moins une fois, alors le temps total d'accès est  $O\left(m + \sum_{i=1}^n q(i) \log\left(\frac{m}{q(i)}\right)\right)$ .

Ce théorème implique qu'un arbre déployé est aussi efficace qu'un arbre de recherche fixé, y compris l'arbre optimal pour la séquence d'accès. En effet, le temps total d'accès pour n'importe quel arbre fixé est  $\Omega\left(m + \sum_{i=1}^n q(i) \log(m/q(i))\right)$ .

On suppose que les éléments sont numérotés de 1 à  $n$  en ordre infixé (fils gauche - racine - fils droit). Soit la séquence d'accès  $i_1, i_2, \dots, i_m$ .

7. Montrer le **théorème du "doigt/pointeur statique"** : si  $f$  est un élément fixé, alors le temps total d'accès est  $O(n \log n + m + \sum_{j=1}^m \log(|i_j - f| + 1))$ .

Ce théorème indique qu'un arbre déployé a la même performance lors de la recherche d'un élément dans le voisinage d'un élément pointé, qu'un arbre de recherche à pointeur (*finger tree*).

Si l'on modifie le poids des éléments pendant les accès on peut obtenir de nouveaux résultats intéressants. Pour tout nouvel accès  $j$ , on note  $t(j)$  le nombre d'éléments différents qui ont été accédés depuis le dernier accès à l'élément  $i_j$ , ou depuis le début de la séquence si  $j$  est le premier accès à l'élément  $i_j$ .

8. Montrer le **théorème de l'ensemble de travail** : le temps total d'accès est  $O(n \log n + m + \sum_{j=1}^m \log(t(j) + 1))$ .

Ce dernier théorème nous dit que le temps d'accès à un élément  $i$  peut être estimé comme étant le logarithme de 1 plus le nombre d'éléments différents accédés depuis le dernier accès à  $i$ .

## Références

- [1] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3) :652–686, 1985.