TD 9 – Analyse amortie

Exercice 1. *Pile*

On considère une pile munie des opérations suivantes :
– $PUSH(S, x)$: empile un objet x sur la pile S
– $POP(S)$: dépile le sommet de la pile S et retourne l'objet dépilé
– $MULTIPOP(S, k)$: dépile au plus k objets de la pile S

Algorithm 1: $MULTIPOP(S, k)$

```
début
    tant que S ≠ ∅ et k ≠ 0 faire
        POP(S);
        k ← k − 1;
```

1. Quelle est la complexité de chacune des 3 opérations ? En déduire avec la méthode globale (méthode de l'agrégat) le coût amorti pour une suite de n opérations $PUSH$, POP et $MULTIPOP$ sur une pile initialement vide.
2. Même question avec la méthode des acomptes.
3. Même question avec la méthode des potentiels.
4. On souhaite implémenter une file à l'aide de deux piles, de telle façon que le coût amorti des opérations $ENQUEUE$ et $DEQUEUE$ soit $O(1)$. Comment peut-on faire ?

Exercice 2. *Structure de donnees*

On souhaite avoir une structure de données S contenant des réels quelconques (pouvant être égaux entre eux). Cette structure doit supporter les deux opérations suivantes :
– $Insertion(S, x)$: insère x dans S
– $SuppressionMoitieSuperieure(S)$: supprime les $\lceil |S|/2 \rceil$ données les plus grandes de S

Expliquer comment implémenter ces deux opérations afin qu'elles s'exécutent en $O(1)$ en temps amorti.

Exercice 3. *Arbres déployés (Splay trees)*

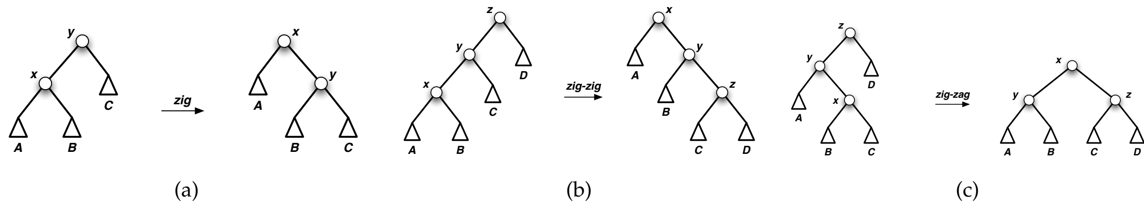
Cet exercice est tiré de l'article [1].

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.



(a) (b) (c)

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.

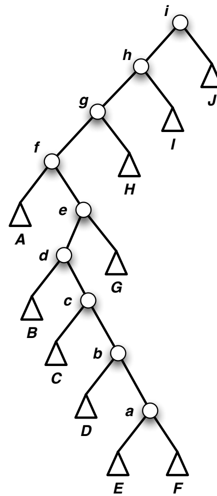


Figure 2 – Exemple pour *déployer*

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 $3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$.

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 $\Omega(m + \sum_{i=1}^{n} q(i)\log(m/q(i)))$.
On suppose que les éléments sont numérotés de 1 à n en ordre infixé (fils gauche - racine - fils droit). Soit la séquence d'accès $i_1, i_2, \ldots, i_m$.

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.