

TD n°12 Analyse Amortie

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

début

 tant que $S \neq \emptyset$ et $k \neq 0$ faire

$POP(S)$;

$k \leftarrow k - 1$;

 fin

fin

Algorithme 1 : $MULTIPOP(S, k)$

Question 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.

Solution : Les opérations $PUSH$ et POP se font en $O(1)$, et $MULTIPOP$ en $O(\min\{|S|, k\})$.

Chaque objet peut être dépilé au plus une fois pour chaque empilement de ce même objet. Donc on peut avoir au plus autant d'appel à POP qu'il y a eu d'appels à $PUSH$, y compris pour les POP appelés au sein de la procédure $MULTIPOP$. On a au plus n appels à $PUSH$. Donc une suite quelconque de n opérations $PUSH$, POP et $MULTIPOP$ aura un temps total de $O(n)$. On a donc un coût moyen par opération de $O(n)/n = O(1)$. Dans l'analyse par agrégat, chaque opération se voit affecter le même coût amorti, donc ici $PUSH$, POP et $MULTIPOP$ ont toutes un coût de $O(1)$. \square

Question 1.2 Même question avec la méthode des acomptes.

Solution : Dans cette méthode, chaque opération peut avoir un coût différent. Lorsque le coût affecté à une opération est supérieur à son coût réel, alors le crédit restant sert à payer les opérations qui ont un coût amorti plus faible que leur coût réel.

On attribut ici un coût amorti 2 pour l'opération $PUSH$, et 0 pour POP et $MULTIPOP$. Les 3 coûts sont en $O(1)$ et on remarque que le coût de $MULTIPOP$ est constant, alors qu'en réalité il est variable.

Lorsqu'une opération $PUSH$ est réalisé on paye 1 euro pour l'opération, et on associe l'euro restant à l'objet ainsi ajouté pour pouvoir payer plus tard l'opération POP correspondante. Lorsqu'on réalise une opération POP on prend l'euro associé à l'objet pour payer l'opération, et on n'a alors pas à payer plus pour payer le véritable coût de l'opération. Il n'est pas nécessaire de payer pour l'opération $MULTIPOP$, puisqu'elle sera payée par les opérations POP correspondantes. On s'assure qu'à chaque instant le nombre d'euros présents dans la pile est positif (on ne retire pas plus que ce qu'on a apporté).

Donc pour une séquence de n opérations, on a un coût amorti total en $O(n)$ qui est bien le même que le coût réel. \square

Question 1.3 Même question avec la méthode des potentiels.

Solution : On définit la fonction potentiel Φ de la pile comme étant le nombre d'objets présents dans la pile. Pour la pile vide $\Phi(D_0) = 0$. Puisque le nombre d'objets dans la pile n'est jamais négatif, on a toujours un potentiel non négatif, et donc $\Phi(D_i) \geq 0 = \Phi(D_0)$.

La différence de potentiel après une opération *PUSH* est $\Phi(D_i) - \Phi(D_{i-1}) = (|S| + 1) - |S| = 1$. D'où un coût amorti de $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.

La différence de potentiel après une opération *POP* est $\Phi(D_i) - \Phi(D_{i-1}) = (|S| - 1) - |S| = -1$. D'où un coût amorti de $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$.

La différence de potentiel après une opération *MULTIPOP*(S, k), avec $k' = \min\{|S|, k\}$ est $\Phi(D_i) - \Phi(D_{i-1}) = -k'$. D'où un coût amorti de $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$.

Le coût amorti de chacune des opérations est $O(1)$, est on a donc un coût total amorti en $O(n)$ pour n opérations. \square

Question 1.4 On souhaite implémenter une file à l'aide de deux piles, de telle façon que le coût amorti des opérations *ENQUEUE* et *DEQUEUE* soit $O(1)$.

Solution : On utilise pour cela deux piles : *PileEnt* et *PileSor*. Lorsqu'un élément est ajouté à la file, il est ajouté dans *PileEnt*. Lorsqu'un élément est retiré de la file, il est retiré de *PileSor*. Si la pile *PileSor* est vide, alors on dépile *PileEnt* et on rempile les éléments dans *PileSor*. On a donc une troisième méthode qui est *TRANSFER*.

On a un coût amorti de 3 pour *ENQUEUE* et de 1 pour *DEQUEUE*. En effet, un élément lorsqu'il est ajouté utilise au plus 2 fois *PUSH* et une fois *POP* (un *PUSH* pour l'ajouter dans *PileEnt*, et un *POP* et un *PUSH* pour l'insérer dans *PileSor* lors d'un transfert) s'il n'est pas sorti par *DEQUEUE*. Pour la sortie il faut réaliser *POP* une fois. \square

2 Structure de données

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.

Solution : On implémente S avec une liste non triée. L'insertion prend donc bien un temps $O(1)$ dans le pire des cas.

La suppression peut être réalisée en $O(|S|)$ dans le pire cas. Tout d'abord trouver la médiane de S en $O(|S|)$ (voir cours). Puis, en $O(|S|)$ parcourir la liste et supprimer les $\lceil |S|/2 \rceil$ éléments qui sont plus grands ou égaux à la médiane.

On définit la fonction potentiel $\Phi(S) = 2|S|$. On a donc les coûts amortis suivants :

- le coût pour l'insertion est de 1, et son coût amorti : $\hat{c} = c + \Delta\Phi \leq 1 + 2(|S| + 1 - |S|) = 3$
- le coût pour la suppression est $|S|$, et son coût amorti : $\hat{c} = c + \Delta\Phi \leq |S| + 2(|S|/2 - |S|) = 0$ \square

3 Splay tree

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.

3.1 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.

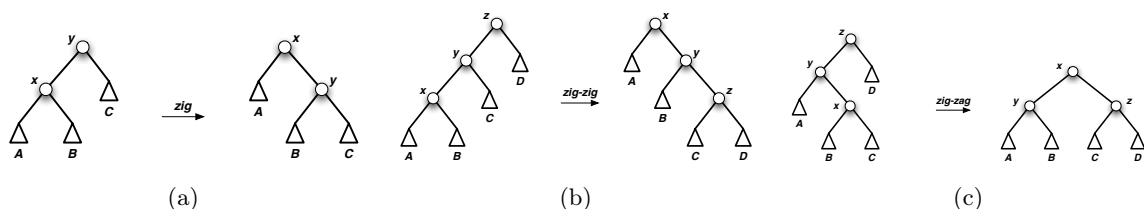


FIG. 1: Les différentes étapes pouvant intervenir durant le déploiement : 1a zig, 1b zig-zig et 1c zig-zag.

Question 3.1 Appliquer *déployer* sur le nœud a de l'arbre figure 2.

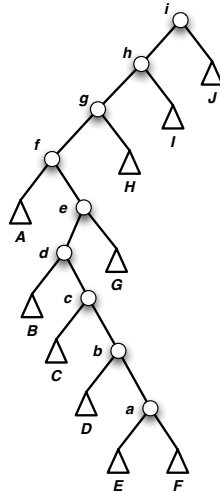


FIG. 2: Exemple pour *déployer*

Solution : Voir figure 3. □

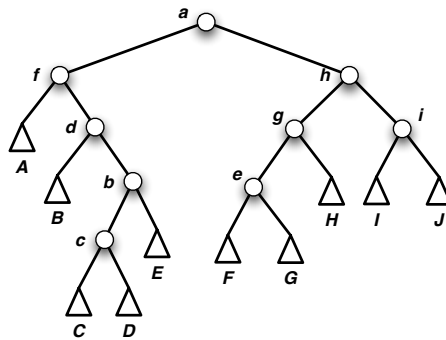


FIG. 3: Solution exemple pour *déployer*

Question 3.2 Quelle est la complexité de l'exécution de l'heuristique *déployer* sur un nœud x ?

Solution : L'application de *déployer* sur un nœud x à une profondeur d demande un temps $O(d)$, c'est à dire un temps comparable à celui pour accéder à l'élément x . □

3.2 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.

Question 3.3 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 $\log(x) + \log(y) \leq -2$.

Remarque : Calculer une borne supérieure "pratique" de chacun des 3 cas de l'heuristique, pour pouvoir calculer facilement le coût amorti de l'exécution de *déployer* (pour *zig* : $1 + 3(r'(x) - r(x))$, et pour *zig-zig* et *zig-zag* : $3(r'(x) - r(x))$).

Solution : Temps amorti a : $a = t + \Phi' - \Phi$, avec t le réel temps d'exécution, Φ le potentiel avant l'opération, et Φ' le potentiel après.

Soient y le père de x , et z le père de y (s'il existe) avant l'opération.

Cas 1 (zig) : une rotation est réalisée, donc le temps amorti de l'opération est :

$$\begin{aligned} 1 + r'(x) + r'(y) - r(x) - r(y) & \text{ puisque seuls } x \text{ et } y \text{ peuvent changer de rang} \\ & \leq 1 + r'(x) - r(x) && \text{ puisque } r(y) \geq r'(y) \\ & \leq 1 + 3(r'(x) - r(x)) && \text{ puisque } r'(x) \geq r(x) \end{aligned}$$

Cas 2 (zig-zig) : deux rotations sont réalisées, donc le temps amorti de l'opération est :

$$\begin{aligned} 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) & \text{ puisque seuls } x, y \text{ et } z \text{ peuvent changer de rang} \\ = 2 + r'(y) + r'(z) - r(x) - r(y) & \text{ puisque } r'(x) = r(z) \\ \leq 2 + r'(x) + r'(z) - 2r(x) & \text{ puisque } r'(x) \geq r'(y) \text{ et } r(y) \geq r(x) \end{aligned}$$

On a $2 + r'(x) + r'(z) - 2r(x) \leq 3(r'(x) - r(x))$, c'est-à-dire $2r'(x) - r'(z) - r(x) \geq 2$. Preuve : la convexité de la fonction \log implique que $\log x + \log y$ lorsque $x, y > 0$ et $x + y \leq 1$, est maximum avec la valeur -2 , pour $x = y = 1/2$. Donc, $r(x) + r'(z) - 2r'(x) = \log(s(x)) + \log(s'(z)) - 2\log(s'(x)) = \log(s(x)/s'(x)) + \log(s'(z)/s'(x)) \leq -2$ puisque $s(x) + s'(z) \leq s'(x)$. Donc $2r'(x) - r(x) - r'(z) \geq 2$. Le cas 2 a donc un coût amorti d'au plus $3(r'(x) - r(x))$.

Cas 3 (zig-zag) : le coût amorti de l'opération est :

$$\begin{aligned} 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) & \text{ puisque seuls } x, y \text{ et } z \text{ peuvent changer de rang} \\ \leq 2 + r'(y) + r'(z) - 2r(x) & \text{ puisque } r'(x) = r(z) \text{ et } r(x) \leq r(y) \end{aligned}$$

Ce coût vaut au plus $2(r'(x) - r(x))$, c'est-à-dire $2r'(x) - r'(y) - r'(z) \geq 2$. Comme pour le cas 2, puisqu'on a $s'(y) + s'(z) \leq s'(x)$ on a bien un coût amorti en $2(r'(x) - r(x)) \leq 3(r'(x) - r(x))$.

□

Question 3.4 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 $3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$.

Solution : Lemme d'accès : s'il n'y a pas de rotation on obtient bien la borne, puisque dans ce cas $t = x$ et que par définition on considère un coût de 1 s'il n'y a pas de rotation.

S'il y a des rotations on calcule la somme du coût amorti de chacune des étapes. Les termes s'annulent pour arriver à un coût d'au plus $3(r'(x) - r(x)) + 1 = 3(r(t) - r(x)) + 1$ (annulation de tous les termes sauf pour le zig final). □

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 $W = O(1)$.

Remarque : Si le poids des éléments est fixé, alors la différence de potentiel sur la séquence est d'au plus $\sum_{i=1}^n \log(W/w(i))$, puisque la taille d'un nœud contenant l'élément i est d'au plus W et d'au moins $w(i)$. On a donc, $\sum \log(w(i)) \leq \text{potentiel} \leq \sum \log(W)$. Donc la différence de potentiel que peut subir chaque nœud est d'au plus $\log(W) - \log(w(i))$, d'où le $\sum_{i=1}^n \log(W/w(i))$.

Question 3.5 Montrer le **théorème d'équilibre** : le temps total d'accès est $O((m+n) \log n + m)$.

Solution : On choisit un poids de $1/n$ pour chaque élément. On a alors $W = 1$, le coût amorti d'accès est d'au plus $3 \log n + 1$ pour chaque élément (soit $3m \log n + m$ pour m accès), et la différence de potentiel sur toute la suite d'accès est d'au plus $n \log n$. On en déduit donc que le coût total d'accès est $O((m+n) \log n + m)$. \square

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.

Question 3.6 Montrer le **théorème d'optimalité statique** : si tous les objets sont accédés au moins une fois, alors le temps total d'accès est $O\left(m + \sum_{i=1}^n q(i) \log\left(\frac{m}{q(i)}\right)\right)$.

Solution : On attribut un poids $q(i)/m$ à chaque élément i . On a donc $W = 1$, le coût amorti d'accès à un élément est $O(\log(m/q(i)))$, et la différence de potentiel sur toute la suite d'accès est d'au plus $\sum_{i=1}^n \log(m/q(i))$. On en déduit donc que le coût total d'accès est $O\left(m + \sum_{i=1}^n q(i) \log\left(\frac{m}{q(i)}\right)\right)$ (chaque élément i est accédé $q(i)$ fois). \square

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, \dots, i_m .

Question 3.7 Montrer le **théorème du "doigt/pointeur statique"** : si f est un élément fixé, alors le temps total d'accès est $O(n \log n + m + \sum_{j=1}^m \log(|i_j - f| + 1))$.

Solution : On attribue un poids de $1/(|i-f|+1)^2$ à l'élément i . On a donc $W \leq 2 \sum_{k=1}^{\infty} 1/k^2 = O(1)$. Le temps amorti du $j^{\text{ème}}$ accès est $O(\log(|i_j - f| + 1))$, et la différence de potentiel sur toute la séquence est $O(n \log n)$, puisque le poids de chaque élément est d'au moins $1/n^2$. On a donc bien en temps d'accès total de $O(n \log n + m + \sum_{j=1}^m \log(|i_j - f| + 1))$. \square

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 .

Question 3.8 Montrer le **théorème de l'ensemble de travail** : le temps total d'accès est $O(n \log n + m + \sum_{j=1}^m \log(t(j) + 1))$.

Solution : On assigne les poids $1, 1/4, 1/9, \dots, 1/n^2$ aux éléments par ordre de premier accès (les éléments accédés en premier ont le plus fort poids). À chaque nouvel accès on redéfinit les poids de la façon suivante. Supposons que le poids de l'élément i à l'accès j soit $1/k^2$. Après l'accès j , on assigne le poids 1 à i_j , et pour tout élément i avec un poids de $1/(k')^2$ avec $k' < k$, on assigne le poids $1/(k'+1)^2$ à i . On permute ainsi les poids entre les différents éléments. De plus, cela garanti qu'à l'accès j le poids de i_j sera de $1/(t(j) + 1)^2$. On a $W = \sum_{k=1}^n 1/k^2 = O(1)$, donc le coût amorti d'un accès j est $O(\log(t(j) + 1))$. La réassignation des poids après un accès augmente le poids de l'élément présent à la racine (puisque déployer remonte l'élément à la racine), et diminue le poids d'autres éléments dans l'arbre. La taille de la racine reste inchangée, mais la taille des autres nœuds peut diminuer. Ainsi, le changement des poids ne peut que faire diminuer le potentiel, et le temps amorti pour réassigner les poids est soit zéro soit négatif.

La différence de potentiel sur la séquence est $O(n \log n)$. Donc on a bien $O(n \log n + m + \sum_{j=1}^m \log(t(j) + 1))$ pour le temps total d'accès. \square

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.