TD 06 – Analyse amortie (corrigé)

(Pile) 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.

☞ 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 $O(1)$.

2. Même question avec la méthode des acomptes.

☞ 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 attribue 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.

3. Même question avec la méthode des potentiels.

☞ 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.

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 ?

☞ 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.

(CptBin) Exercice 2. *Remise à zéro d'un compteur binaire*

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 $O(nk)$.

☞

Algorithm 2: Incrementer(A)

```
début
    i ← 0;
    tant que i < |A| et A[i] = 1 faire
        A[i] ← 0;
        i ← i + 1;
    si i < |A| alors
        A[i] ← 1;
```

Pareil pour décrémenter. On a un coût par opération en $O(k)$ soit pour n opération un coût dans le pire cas de $O(nk)$.

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.

☞ On introduit une nouvelle variable $max[A]$ qui contient l'index du 1 de poids fort. Initialement $max[A]$ vaut -1, puisqu'il n'y a pas de 1 dans le compteur. Il est ensuite mis à jour au fur et à mesure de l'incrément, et remis à -1 lorsque le compteur est remis à 0.

Algorithm 3: Incrementer(A)

```
début
    i ← 0;
    tant que i < |A| et A[i] = 1 faire
        A[i] ← 0;
        i ← i + 1;
    si i < |A| alors
        A[i] ← 1;
        si i > max[A] alors
            max[A] ← i;
    sinon
        max[A] ← −1;
```

Algorithm 4: RÀZ(A)

```
début
    pour i ← 0 à max[A] faire
        A[i] ← 0;
    max[A] ← −1;
```

On paye 4 pour Incrémenter(A), et 1 pour RÀZ(A). On suppose que cela nous coûte 1 pour changer un bit, et 1 pour mettre à jour $max[A]$. On paye 1 pour passer un bit à 1, et on place 1 sur ce bit comme crédit pour pouvoir payer la remise à 0 du bit lors d'un incrément. On paye en plus 1 pour mettre à jour $max[A]$ avec l'index du bit de poids fort (si $max[A]$ n'a pas besoin d'être mis à jour tant pis on ne ce sert pas de cet argent). Puisque RÀZ ne manipule que les bits jusqu'à la position $max[A]$, et que chaque bit a au moins une fois été en position de bit de poids fort, alors ils ont tous dessus au moins 1 pour pouvoir payer leur mise à 0. Il nous reste alors juste à payer 1 pour modifier $max[A]$ à -1.

On est donc bien en $O(n)$.

(RechercheElements) Exercice 3. *Recherche d'Elements*

On veut une structure de données qui permette la recherche (d'un élément), et l'insertion (d'un nouvel élément) de manière efficace.

1. Quels sont les coûts de la recherche et de l'insertion si on utilise un tableau trié de taille n ?

☞ Recherche en $\log(n)$, insertion en $O(n)$.

On propose la solution suivante : soit $k = \lceil \log(n+1) \rceil$ et soit $(n_{k-1}, n_{k-2}, \ldots, n_0)$ la représentation binaire de n. On a k tableaux triés $A_0, A_1, \ldots, A_{k-1}$, et la taille de A_i est 2^i pour tout i. Le tableau A_i est plein si $n_i = 1$ et vide si $n_i = 0$. Ainsi le nombre total d'éléments stockés dans les k tableaux est $\sum_{i=0}^{k-1} n_i 2^i = n$. Noter que chaque tableau est trié mais qu'il n'y a aucune relation entre les éléments de deux tableaux.

2. Expliquer comment faire une recherche dans cette structure, et donner le coût au pire cas.

☞ On peut effectuer une recherche dans cette structure de données en répétant la recherche dans chacun des sous-tableaux. La recherche dans un sous-tableau de taille m se fera en $\log(m)$ puisque les sous-tableaux sont triés. au pire cas (tout les tableaux sont pleins et la recherche est infructueuse), le temps total sera :

$$\begin{aligned}
T(n) &= \Theta(\log(2^{k-1}) + \log(2^{k-2}) + \cdots + \log(2^1) + \log(2^0)) \\
&= \Theta((k-1) + (k-2) + \cdots + 1 + 0) \\
&= \Theta(k(k-1)/2) \\
&= \Theta(\lceil \log(n+1) \rceil (\lceil \log(n+1) \rceil - 1)/2) \\
&= \Theta(\log^2(n))
\end{aligned}$$

3. Expliquer comment faire une insertion dans cette structure, et donner le coût au pire cas et en analyse amortie.

☞ On commence par créer un nouveaux tableau trié contenant uniquement l'élément à insérer. Si le tableau A_0 est vide, on remplace A_0 par le nouveau tableau, sinon on crée un nouveau tableau trié de taille deux à partir de A_0 et du nouveau tableau.si A_1 est vide, on remplace A_1 par le nouveau tableau sinon *etc.* Au final, on obtient toujours un tableau A_i de taille 2^i. Au pire cas (les $k-2$ premiers tableaux sont pleins) Le temps de calcul traitement sera :

$$\begin{aligned}
T(n) &= 2(2^0 + 2^1 + \cdots + 2^{k-2}) && (1) \\
&= 2(2^{k-1} - 1) && (2) \\
&= 2^k - 2 && (3) \\
&= \Theta(n) && (4)
\end{aligned}$$

En utilisant la méthode de l'agrégat pour calculer le coût total d'une série de n insertion à partir d'une structure de données vide. Soit r la position du 0 le plus à gauche de la repésentation binaire $(n_{k-1}, n_{k-2}, \cdots, n_0)$ de n. $n_j = 1$ pour $j < r$. Le coût d'une insertion quand n éléments ont déjà été insérés est $\sum_{j=0}^{r-1} 2 \cdot 2^j = \mathcal{O}(2^r)$.

$r = 0$ une fois sur deux, $r = 1$ une fois sur quatre *etc.* et il y a au plus $\lceil n/2^r \rceil$ insertion pour chaque valeur de r. on peut donc borner le coût de n insertions par :

$$\mathcal{O}\left(\sum_{r=0}^{\lceil \log(n+1) \rceil} \left(\left\lceil \frac{n}{2^r} \right\rceil \right) 2^r \right) = \mathcal{O}(n \log(n))$$

Le coup amortie d'une insertion est donc $\log(n)$.

4. Expliquer comment supprimer un élément.

☞ pour supprimer l'élément x :

— Trouver de plus petit j tel que A_j est plein. soit y le plus petit élement de A_j.

— Trouver A_i, le tableau contenant x.

— Enlever x et le remplacer par y. A_j a désomais $2^j - 1$ élément, On place alors le premier élément de A_j dans A_0 les deux suivant dans A_1 *etc.* et on marque A_j comme vide. Il n'y a pas besoin de trier les tableaux nouvellement créés.

(StructureDonnees2) Exercice 4. *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.

(*Indice :* Vous pouvez supposer que vous savez comment calculer la médiane en temps linéaire.)

☞ 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 : $\hat{c} = c + \Delta\Phi \leq |S| + 2(|S|/2 - |S|) = 0$