
UE NFP 136 (Responsable : Cédric Bentz)
Tous documents autorisés

Examen de l'UE NFP 136 (VARI 2) 16 juin 2014

Exercice 1 : détermination du deuxième plus petit élément

Dans cet exercice, on s'intéresse à la façon de déterminer le deuxième plus petit élément d'un tableau d'entiers. Plus précisément, on cherche à implémenter la méthode JAVA suivante :

```
public static int deuxiemePlusPetitElement(int[] tab) {  
    /* corps de la méthode */  
}
```

L'entier renvoyé par cette méthode est précisément la valeur du deuxième plus petit élément du tableau d'entiers `tab` passé en paramètre. On supposera ici qu'un même entier ne peut pas apparaître deux fois dans `tab` (en d'autres termes, `tab` ne contient que des entiers différents).

On considérera trois cas : quand `tab` est un tableau quelconque, quand `tab` est un tableau trié, et quand `tab` est un tas.

Travail demandé

Dans les questions qui suivent, on vous demande d'implémenter cette méthode `deuxiemePlusPetitElement` dans les différents cas suivants :

1. Lorsque `tab` est un tableau d'entiers supposé quelconque,
2. Lorsque `tab` est un tableau d'entiers supposé trié (selon l'ordre croissant des valeurs des éléments),
3. Lorsque `tab` est un tableau d'entiers supposé trié (selon l'ordre décroissant des valeurs des éléments),
4. Lorsque `tab` est la représentation, sous forme d'un tableau d'entiers (et donc pas d'un objet de la classe `Tas` vue en cours !), d'un tas *min*.

À chaque fois, écrivez le code JAVA correspondant, et détaillez la complexité au pire cas de la méthode obtenue (dans chacun de ces cas, on cherchera bien sûr à obtenir la meilleure complexité possible).

Exercice 2 : concaténation de deux listes chaînées

Dans cet exercice, on vous demande d'implémenter une méthode JAVA permettant de concaténer (c'est-à-dire de mettre bout à bout) deux listes chaînées d'entiers (instances de la classe Liste en JAVA). Par exemple, si la première liste contient les cinq entiers suivants

-> 1 -> 4 -> 2 -> 9 -> 5

et si la deuxième liste contient les quatre entiers suivants

-> 2 -> 7 -> 5 -> 3

alors la liste résultant de la concaténation de la première liste avec la deuxième sera la suivante :

-> 1 -> 4 -> 2 -> 9 -> 5 -> 2 -> 7 -> 5 -> 3

Travail demandé

1. Implémentez en JAVA une méthode de la classe Liste, notée `public static Liste concatener(Liste liste1, Liste liste2)`, qui permet de concaténer la liste `liste1` avec la liste `liste2` (toutes deux passées en paramètres), puis de retourner la liste qui en résulte.
2. On note respectivement n_1 et n_2 les tailles (nombres d'éléments) des listes `liste1` et `liste2`, au moment de l'appel de la méthode. Détaillez la complexité en temps (au pire cas) et en espace de votre méthode, en fonction de n_1 et n_2 . Quelle influence a n_1 sur cette complexité en temps ? Et qu'en est-il pour n_2 ?

Exercice 3 : représentations d'un graphe

Dans cet exercice, on vous demande d'implémenter une méthode JAVA permettant de calculer la représentation par matrice d'adjacence d'un graphe orienté à partir de sa représentation par listes d'adjacence.

L'en-tête d'une telle méthode est le suivant :

```
public static void conversionGraphe(int[][] matAdj, Liste[] tabListeAdj) {  
    /* corps de la méthode */  
}
```

Le paramètre `matAdj` est donc destiné à contenir la représentation par matrice d'adjacence du graphe représenté par le tableau de listes d'adjacence `tabListeAdj`. Si n désigne le nombre de sommets du graphe, `matAdj` est donc

une matrice de n lignes et n colonnes, et `tabListeAdj` est un tableau de taille n , dont chaque élément est une liste chaînée (éventuellement vide).

Ainsi, on rappelle que, pour tout couple de sommets i et j , l'élément `matAdj[i - 1][j - 1]` vaudra 1 s'il existe un arc entre les sommets i et j , et 0 sinon. De même, pour tout sommet i , l'élément `tabListeAdj[i - 1]` est la liste chaînée des successeurs de i . Par exemple, si le sommet 3 a pour successeurs les sommets 1 et 4, alors `tabListeAdj[2]` sera représenté par la liste suivante :

-> 1 -> 4

Travail demandé

Implémentez la méthode `conversionGraphe` en JAVA, et déterminez sa complexité en temps au pire cas (on notera n le nombre de sommets du graphe, et m son nombre d'arcs).

Exercice 4 : la plus grande moitié d'un tas

Dans cet exercice, on se propose d'implémenter la méthode JAVA suivante :

```
public static int[] plusGrandeMoitie(int n, Tas unTas) {  
    /* corps de la méthode */  
}
```

Cette méthode prend en paramètres un entier n et un tas *min* d'entiers, appelé `unTas` et que l'on supposera (pour simplifier) de taille n paire, et retourne un tableau d'entiers (de taille $\frac{n}{2}$) contenant les $\frac{n}{2}$ plus grands éléments de `unTas`, triés par ordre croissant. Par exemple, supposons que le tas `unTas` (de taille $n = 6$) soit représenté par le tableau suivant :

1	2	12	10	25	17
---	---	----	----	----	----

Alors, le tableau retourné (de taille $\frac{n}{2} = 3$) doit contenir les trois entiers 12, 17 et 25 dans cet ordre (ordre croissant).

Travail demandé

1. Implémentez la méthode `plusGrandeMoitie` en JAVA, en cherchant à garantir la meilleure complexité en temps possible. Pour cela, vous pouvez utiliser toute méthode de la classe JAVA `Tas` (vue en cours) que vous jugerez nécessaire (`minimum`, `supprimerMin` ou `insérer`).
2. Déterminez la complexité en temps (au pire cas) de cette méthode.
3. **Détaillez toutes les étapes** de l'exécution de cette méthode sur le tas à 6 éléments donné ci-dessus.