Conservatoire National des Arts et Métiers
École SITI - Département Informatique

Année 2016–2017

UE NFP 136 (Responsable : Cédric Bentz)
Calculatrices graphiques et appareils électroniques interdits — Durée : 3h
Documents autorisés : documents papier uniquement

Examen de session 1 de l'UE NFP 136 (VARI 2)

20 juin 2017

Exercice 1 : tri par tas d'une liste chaînée

Dans cet exercice, on souhaite trier, par ordre décroissant, une liste chaînée contenant n entiers donnés, à l'aide d'un tri par tas. Pour cela, on pourra utiliser, sans la réécrire, toute méthode de la classe JAVA Tas (vue en cours) qui paraîtra nécessaire (`minimum`, `supprimerMin` ou `inserer`).

Par exemple, si la liste chaînée à trier contient les 10 entiers suivants :

```
-> 7 -> 3 -> 2 -> 5 -> 7 -> 1 -> 4 -> 3 -> 6 -> 8
```

Alors, on obtient à la fin la liste chaînée suivante, à l'aide d'un tas min :

```
-> 8 -> 7 -> 7 -> 6 -> 5 -> 4 -> 3 -> 3 -> 2 -> 1
```

Travail demandé

1. Implémenter cette méthode en JAVA. On précise que son en-tête sera : `public static Liste triDecListeParTasMin(Liste listeATrier)`, où `listeATrier` est la liste chaînée contenant les n entiers à trier, et la valeur retournée est la liste chaînée contenant les n entiers de `listeATrier` triés **par ordre décroissant**.

2. Donner la complexité en temps (au pire cas) et en espace de la méthode `triDecListeParTasMin`, en fonction du nombre d'entiers à trier n.

3. Illustrer en détails le déroulement de `triDecListeParTasMin`, étape par étape, sur la liste de taille 10 donnée ci-dessus en exemple.

Exercice 2 : calcul du nombre de prédécesseurs de tous les sommets dans un graphe orienté

Dans cet exercice, on considère un graphe orienté, qui sera donné soit sous la forme d'une matrice d'adjacence, soit sous la forme d'un tableau de listes d'adjacence. On s'intéresse en particulier à l'ensemble des **prédécesseurs** d'un sommet s donné, c'est-à-dire l'ensemble des sommets p tels qu'il existe un arc (p, s). Dans ce cas, on dit aussi, de façon symétrique, que le sommet s est un **successeur** du sommet p.

Par exemple, si on se donne un graphe orienté contenant 4 sommets $1, 2, 3, 4$ et 6 arcs $(1, 2), (1, 3), (1, 4), (2, 3), (4, 2), (4, 3)$, alors :

- le sommet 1 n'a aucun prédécesseur,
- le sommet 2 a 2 prédécesseurs (1 et 4),
- le sommet 3 a 3 prédécesseurs $(1, 2, 4)$,
- le sommet 4 a 1 seul prédécesseur (1).

On rappelle que, si n désigne le nombre de sommets d'un graphe orienté, la matrice d'adjacence de ce graphe est une matrice d'entiers contenant n lignes et n colonnes telle que, pour tout couple de sommets i et j, l'élément sur la ligne $i - 1$ et la colonne $j - 1$ vaut 1 s'il existe un arc du sommet i vers le sommet j (autrement dit si j est un **successeur** de i), et 0 sinon.

Par ailleurs, le tableau de listes d'adjacence d'un tel graphe est un tableau de taille n, dont chaque élément (l'élément d'indice $i - 1$ étant associé au sommet i) est une liste chaînée d'entiers (éventuellement vide), contenant, pour chaque i, les numéros des **successeurs** du sommet i.

Travail demandé

Implémenter en JAVA une méthode renvoyant le nombre de prédécesseurs de chaque sommet d'un graphe orienté, dans le cas où ce graphe est donné :

1. sous la forme d'une matrice d'adjacence ; l'en-tête de la méthode sera alors `public static int[] nbPredecesseurs(int[][] matAdj)`, où le tableau d'entiers renvoyé contiendra le nombre de prédécesseurs de chaque sommet du graphe, et le paramètre `matAdj` est la matrice d'adjacence du graphe considéré,
2. puis sous la forme d'un tableau de listes d'adjacence ; l'en-tête de la méthode deviendra alors `public static int[] nbPredecesseurs (Liste[] tabListesAdj)`, où le tableau d'entiers renvoyé sera défini comme ci-dessus, et le paramètre `tabListesAdj` est le tableau de listes d'adjacence du graphe considéré.

Dans les deux cas, on déterminera la complexité en espace et la complexité en temps au pire cas (on notera n le nombre de sommets du graphe).

Exercice 3 : insertion dans une liste chaînée après la dernière occurrence d'un élément donné

Dans cet exercice, on vous demande d'implémenter une méthode JAVA permettant d'insérer, dans une liste chaînée d'entiers donnée, un élément (entier) x donné, juste après la **dernière** occurrence d'un autre élément (entier) y donné. Ainsi, si $x = 5$ et $y = 4$, et si la liste considérée est comme ceci

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

alors la liste que l'on souhaite obtenir à la fin est la suivante :

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

Travail demandé

1. Implémenter en JAVA cette méthode de la classe Liste, dont l'en-tête est : `public static void insererApresDernier(Liste l, int x, int y)`, où x et y sont définis comme ci-dessus, et l est la liste dans laquelle insérer l'élément x. À noter : si l'élément y n'apparaît pas dans la liste, alors cette méthode ne fera rien.
2. Détailler la complexité en temps (au pire cas) et en espace de cette méthode, en fonction de la taille n de la liste l.

Exercice 4 : nombre d'éléments plus petits dans un tableau

Dans cet exercice, on vous demande d'implémenter une méthode JAVA, dont l'en-tête est `public static int nombrePlusPetits(int[] t, int i)`, et qui renvoie le nombre d'éléments d'un tableau d'entiers `t` de taille n (ces n entiers étant tous différents) qui sont plus petits que l'élément d'indice `i` donné dans `t`. Par exemple, si `t` contient les 8 entiers suivants

```
|19|11|2|5|17|7|3|13|
```

et si `i` $= 5$ (indice de l'élément de valeur 7), alors la méthode renverra 3.

Travail demandé

Implémenter cette méthode en JAVA, en garantissant à chaque fois la complexité en temps au pire cas la plus faible possible, **qu'on précisera** :

1. Dans le cas où les éléments de `t` sont déjà triés par ordre croissant.
2. Dans le cas où les éléments de `t` sont déjà triés par ordre décroissant.
3. Dans le cas où les éléments de `t` sont dans un ordre quelconque.

Exercice 5 : système de gestion de fichiers Unix

On considère un disque qui abrite physiquement une partition Unix.

Le système de gestion de fichiers Unix structure les fichiers en blocs de 2048 octets. Un numéro de blocs occupe 4 octets.

On s'intéresse dans la suite à un fichier Unix de 32 Mo.

Travail demandé

1. Combien de blocs de données comprend ce fichier ?
2. Combien d'entrées comporte un bloc d'index ?
3. Dessiner la structure représentant le fichier.
4. Combien de blocs d'index comporte le fichier ?