Université Paris Diderot — Année 2017-2018
L1 - Informatique — 8h30 - 10h30

Partiel IP2 Vendredi 16 mars 2018

Aucun document autorisé

Ce sujet est composé de 4 exercices traitant chacun une partie de ce que nous avons vu ensemble. Ils sont indépendants, et à mon avis de difficulté croissante. La stratégie la plus efficace pour cette épreuve consiste à répondre très proprement aux questions que vous aborderez en premier, ce qui devrait vous assurer la moyenne. Vous pourrez ensuite gérer le temps qu'il vous reste sur ce qui vous semble difficile. Le barème est indicatif.

Exercice 1 (4 points) (Petite modélisation) Un binôme est un objet qui se caractérise par une paire de chaînes de caractères. Ils sont numérotés de façon unique et automatiquement à leur création. Leur représentation est normalisée dans le sens où chaque paire a toujours sa première chaîne plus petite ou égale dans l'ordre lexicographique que sa seconde chaîne (c.à d. que dans un dictionnaire la première chaîne apparaîtrait avant la seconde). La constitution de ces binômes pourra changer avec le temps. On souhaite également conserver un représentant des plus petits binômes qui ont déjà été créés : un binôme b_1 est strictement plus petit qu'un autre binôme b_2, si la première chaîne de b_1 est plus petite strictement que la première de b_2, ou, dans le cas où celles ci sont égales, si la seconde chaîne de b_1 est strictement plus petite que celle de b_2.

Ecrivez une classe `Binôme` en justifiant vos choix. Elle devra contenir entres autres :

- une méthode `normalise` qui assure la représentation normalisée
- une méthode `compare` qui permet de comparer l'ordre lexicographique

Précisez bien la signature de ces méthodes, de toutes vos méthode auxiliaires s'il y en a, et de tous les attributs introduits.

(Rappel : la classe String contient une méthode `char charAt(int index);`)

Exercice 2 (5.5 points) Nous avons vu que sur les listes les opérations qui changent la structure doivent être écrites avec précaution. On considère ici une classe `Cell` de cellules simplement chaînées d'éléments de type E, et une classe `Liste` possédant un lien vers le début de la séquence de cellules.

L'opération qui nous intéresse consiste à extraire de la liste la tranche des cellules comprises entre les indices i (inclus) et j (inclus). On rappelle que les indices commencent à 0. Vous ferez en sorte d'avoir une implémentation robuste des paramètres : c.à d. d'envisager les cas où les index sont improbables.

1. **(0.5 points)** Ecrivez une partie de la classe `List` : la déclation des attributs, un constructeur de liste vide, et un constructeur prenant en argument une cellule
2. **(0.5 points)** Ecrivez une partie de la classe `Cell` : la déclaration des attributs, le getter et setter pour obtenir au fixer la cellule suivante.
3. **(1 point)** Ecrivez une méthode (dans la/les classe de votre choix) qui retourne la cellule d'indice i si elle existe `null` sinon
4. **(0.5 point)** Ecrivez une méthode (dans la/les classe de votre choix) qui retourne la cellule précédent celle d'indice i si elle existe `null` sinon
5. **(3 points)** Ecrivez une méthode **extrait(...)** de la classe `Liste` qui modifie la liste courante en extrayant les cellules d'indices entre i et j (indices compris), et retourne la liste extraite (éventuellement vide). Il y a plusieurs cas limites à prendre en compte, illustrez les très méthodiquement. La liste originale est également modifiée, elle ne contient plus la partie extraite.

Exercice 3 (4.5 points) On s'intéresse à des objets dont le seul attribut est un tableau de listes d'entiers, et on souhaite compter le nombre d'occurrences d'une valeur x dans toute la structure (c.à d. le nombre de fois où elle apparaît. x est un paramètre de la méthode)

Ecrivez ce qui vous sera nécessaire (définition des classes, méthodes appropriées) mais en n'utilisant aucun `for`, ni aucun `while` : **cet exercice doit être résolu dans un style purement récursif**. Il vous faudra peut être introduire des fonctions auxiliaires.

Exercice 4 (6 points) Un petit jeu sur les mots consiste à rogner leurs parties semblables en un certain sens, et à en produire un nouveau. Ainsi avec pour premier mot *'émancipé'* et pour second *'épicés'* on produira *'émanés'*, dans une opération où la dernière lettre du premier mot et la première lettre du second s'annulent tant que c'est possible. Dans un autre exemple : *'électroménager'* rogné par *'regain'* donne *'électroménin'* car l'une après l'autre les premières lettres de *'regain'* : 'r' puis 'e' puis 'g' puis 'a' effacent la fin d'*'électroménager'*. Les parties restantes sont accolées. *'mobile'* rogné par *'elle'* donne *'mobile'* à nouveau, c'est une coïncidence. Bien souvent les mots sont justes juxtaposés : *'auto'* rogné par *'bus'* donne *'autobus'* car aucun effacement n'a lieu.

Le but de cet exercice est de mettre en place la création de ce mot rogné dans un contexte où les mots sont modélisés par des listes chaînées de simples caractères.

- L'implémentation exacte des listes est laissée à votre appréciation, mais n'utilisez pas de tableaux ni de `String`. On cherche bien entendu une certaine efficacité. Vous appellerez vos classes `ListeChar` et `Cellchar`.
- La méthode finale demandée est

```
// a la fin this et mot2 contiennent le meme mot resultat
// et re-utilisent les cellules d'origine
public void produitRogne(ListChar mot2);
```

- Toutes les méthodes qui vous sont utiles doivent être présentées dans la classe appropriée.
- Il est très probable que vous aurez besoin d'écrire entre autres : `concat, removeFirst, removeLast` Faites le proprement.