Université Paris 7 - Denis Diderot — IF2 : Structures de données et Objets JAVA
L1 Sciences — Année 2009-2010, 2ème semestre

TP n°5

Récursivité-Structure de Liste

Exercice 1 ***Suite de Syracuse.*** La suite de Syracuse est définie par la relation suivante :

$$\begin{cases} u_0 & > 0 \\ u_{n+1} & = u_n/2 \quad \text{si } u_n \text{ est pair} \\ u_{n+1} & = 3 * u_n + 1 \quad \text{si } u_n \text{ est impair} \end{cases}$$

Dans une classe `Syracuse` :

1. Écrire une méthode récursive `public static int syracuse(int n,int init)` qui calcule le n$^{\text{ième}}$ terme de la suite de Syracuse telle que $u_0 =$ `init`.
2. Écrire le code suivant dans une méthode :
```
int s, n=0;
while((s = syracuse(n,init)) != 1){
  System.out.println(s);
  n++;
}
```
Quel est le rôle de l'instruction `((s = syracuse(n,init)) != 1)` ? Donnez la valeur que vous voulez à `init`. Quelle est la condition d'arrêt de ce programme ? Pour quelles valeurs d'entrée s'arrête-t-il ? Pouvez vous *montrer* qu'il s'arrête pour toutes les valeurs d'entrée ?
3. Le programme précédent n'est pas optimal : en effet, lors de l'appel de `syracuse(n+1,init)`, on oublie que l'on a déjà calculé précédemment `syracuse(n,init)`. Définissez une méthode statique récursive `void syracuseStop(int terme)` qui :
   – affiche à l'écran le terme courant.
   – calcule le terme suivant de la suite de syracuse (en fonction du terme précédent passé en argument).
   – s'arrête si le terme courant vaut `1` et appelle `syracuseStop(termeSuivant)` sinon.
   Testez le ensuite.
4. En fait, on sait (par des démonstrations informatiques) que pour (au moins) tout $u_0 < 2^{62}$, il existe un rang n tel que $u_n = 1$. La méthode que vous avez programmée termine-t-elle pour toute valeur passée en argument ? Et si les entiers sont codés sur `64` bits ?

Exercice 2 On veut implémen v(t)3.88d4.8850933(e)-444(m)-2.66764(o)5(n)-5.44495(s)-1.55762(t)3.88733(r)

– `first` : retourne le premier élément de la liste.
– `last` : retourne le dernier élément de la liste.
– `next` : retourne l'élément suivant l'élément courant dans la liste.
– `previous` : retourne l'élément précédent l'élément courant dans la liste.
– `find` : cherche si un élément est présent dans la liste.

On impose de plus que la liste soit à accès séquentiel : elle ne peut être parcourue que séquentiellement, c'est-à-dire de proche en proche. On ne peut pas se rendre directement à une position donnée. Cela signifie qu'il y a toujours un élément courant dans la liste à partir duquel on peut avancer ou reculer.

On choisit d'implémenter cette structure de données par une classe `ListeDEntiers` qui stocke les éléments de la liste dans un tableau `elements`. Le fait que la liste soit stockée dans un tableau ne doit pas apparaître depuis les autres classes.

Les éléments étant stockés dans un tableau dont la taille est fixe, comment peut-on gérer l'ajout et la suppression d'un élément ?

On propose d'ajouter une propriété `nombreElements` qui contient le nombre d'éléments de la liste. La taille du tableau `elements` et le nombre d'élément de la liste `nombreElements` sont donc deux valeurs différentes. Lorsqu'on supprime un élément, on ne modifie pas la taille du tableau mais on décrémente la propriété `nombreElements`.
Lorsqu'on ajoute un élément, il faut incrémenter la propriété `nombreElements` et éventuellement augmenter la taille du tableau `elements`. Dans ce dernier cas, la seule solution est de créer un nouveau tableau d'une taille supérieure et de recopier tous les éléments existants dans ce dernier.

1. Créer la classe `ListeDEntiers` avec les propriétés `elements` et `nombreElements`. Pour enregistrer la position de l'élément courant, on ajoute une propriété `curseur` de type entier.
2. Écrire un constructeur sans paramètre qui crée une liste vide. La taille du tableau `elements` sera de 50. Comment initialiser le curseur ?
3. Écrire un constructeur avec un paramètre de type tableau et qui crée une liste contenant les éléments de ce tableau dans le même ordre. La taille du tableau `elements` sera égale Ã 1,5 fois celle du tableau transmis en paramètre. Le curseur sera initialisé à 0 (au début, l'élément courant est le premier de la liste).
4. Redéfinir la méthode `toString` pour qu'elle affiche la longueur et les éléments de la liste. L'élément courant sera précédé d'un `C`.
   NB : *Pour chacun des exercices suivant, tester sur des exemples la classe `ListeDEntiers` depuis une classe `Test`.*
5. Écrire les méthodes `isEmpty` et `length`.
6. Écrire une méthode `current` qui retourne la valeur de l'élément courant. Y-a-t-il une précondition ?
7. Écrire les méthodes `first` et `last`. La méthode `first` positionne l'élément courant au premier élément de la liste et retourne sa valeur. La méthode `last` fait la même chose avec le dernier. Quelles sont les préconditions de ces deux méthodes ?
8. Écrire une méthode `find` qui cherche si une valeur est présente dans la liste. Elle retourne vrai si c'est le cas et positionne le curseur sur la première occurrence de la valeur, sinon elle retourne faux.

9. Écrire la méthode `removeAll`. On fixe alors la taille du tableau `elements` à 50. Que faire du curseur ?
10. Écrire la méthode `remove` qui supprime l'élément courant. Que faire des éléments qui suivent celui-ci ? Le curseur est positionné sur l'élément qui suit l'élément supprimé. S'il y en a pas, il reste sur le dernier élément.
11. Écrire la méthode `add` qui ajoute un élément à la fin de la liste.
    Pour augmenter la taille du tableau lorsque cela est nécessaire, on fera appel à une méthode privée `allonger`. Cette méthode doit augmenter la taille du tableau `elements` de moitié. On pourra aussi créer une méthode privée booléenne `isFull`.
12. Écrire la méthode `insertAfter` qui insère un nouvel élément fourni en paramètre après l'élément courant. Quel comportement choisir si la liste est vide ?
13. Écrire deux méthodes booléennes `hasNext` et `hasPrevious` qui retournent vrai s'il existe respectivement un successeur ou un prédécesseur à l'élément courant, faux sinon.
14. Écrire les méthodes `next` et `previous`. Elles retournent respectivement le successeur ou le prédécesseur et positionne le curseur sur celui-ci.
    Quelles sont les préconditions de ces méthodes ?
15. Écrire une méthode `bubbleSort` qui trie les éléments par l'algorithme du tri à bulle. Le curseur doit rester positionné sur le même élément.