Grands Moulins

INITIATION À LA PROGRAMMATION (IP2)

TP6 bis - Interprète Brainf*ck

L'objectif de ce TP est de manipuler des listes doublement chaînées, c'est-à-dire des structures de liste dans lesquelles chaque cellule possède non seulement une référence vers la cellule suivante mais aussi une référence vers la cellule précédente.

Présentation de Brainf*ck

Brainf*ck est un langage « exotique » inventé par Urban Müller en 1993. Ce langage est voulu très simple : il ne possède que 8 instructions et son modèle d'exécution est très proche de celui des *machines de Turing*. Ce modèle utilise un tableau de 30 000 octets initialisés à 0, et un pointeur sur l'octet courant. Le tableau suivant donne la syntaxe et la sémantique des instructions disponibles.

`>`	incrémente le pointeur
`<`	décrémente le pointeur
`+`	incrémente l'octet courant
`-`	décrémente l'octet courant
`.`	affiche l'octet courant
`,`	demande la valeur de l'octet courant à l'utilisateur
`[`	saute à l'instruction après le `]` correspondant si l'octet courant est à 0
`]`	retourne au `[` correspondant

Les programmes écrits en Brainf*ck peuvent être remarquablement concis mais également incroyablement illisibles. Par exemple, le programme suivant demande deux valeurs à l'utilisateur, les écrit côte-à-côte dans la mémoire et les additionne. À la fin de l'exécution, le programme affiche le résultat de l'addition, qui se retrouve à la position de la première valeur. La seconde valeur se retrouve à 0.

```
,>,[-<+>]<.
```

Dans cette séance, nous allons représenter le tableau et son pointeur par une liste doublement chaînée dont les valeurs seront de simples entiers. Le but est d'écrire un interprète pour le langage capable d'exécuter des programmes Brainf*ck.

Représentation de la mémoire

1. Créez une classe `Memoire` contenant 3 attributs privés : `precedente` et `suivante` de type `Memoire`, et un entier `valeur`.
2. Donnez un premier constructeur à la classe `Memoire` qui ne prend aucun paramètre, initialise les deux références à `null` et `valeur` à 0.
3. Donnez un second constructeur à `Memoire`, qui prend la taille de la mémoire en paramètre. La case mémoire à construire doit être la première.

Manipulation de la mémoire

4. Écrivez des *getters* pour les 3 attributs privés, et un *setter* pour `valeur`.
5. Pour tester vos méthodes, écrivez une méthode `inspecte` qui permet d'afficher la totalité du contenu de la mémoire, ainsi que d'indiquer la position de la case mémoire courante. Par exemple dans la méthode `main` de votre classe de test, le code suivant à gauche doit produire la sortie à droite.

```
Memoire m = new Memoire(5);
m.inspecte();
m = m.getSuivante().getSuivante().getSuivante();
m.setValeur(42);
m.inspecte();
m = m.getPrecedente();
m.setValeur(11);
m.inspecte();
```

```
---------
0 0 0 0 0
^
---------
----------
0 0 0 42 0
      ^
----------
-----------
0 0 11 42 0
    ^
-----------
```

Indication : vous pouvez écrire une méthode privée responsable de l'affichage des cases mémoires à gauche de la case courante (les cases *précédentes*), et une autre responsable de l'affichage des cases à droite (les *suivantes*).
Astuce : pour afficher les espaces et les tirets, pensez à la méthode `repeat(int)` de la classe `String` (consultez l'API).

Implémentation de l'interprète

6. Dans votre classe de test, écrivez une méthode statique `brainf_ck(String programme)` qui prend en paramètre une chaîne de caractère représentant le programme Brainf*ck à exécuter. Déclarez une mémoire de la taille que vous voulez (pas trop grande pour pouvoir afficher son contenu si vous en avez besoin). Utilisez une boucle pour parcourir les caractères du programme, ainsi qu'un `switch` pour tester le caractère courant, avec un `case` pour chacune des 8 instructions possibles. Pour l'instant, laissez simplement un `break` dans chacune des branches, nous allons les remplir progressivement.
7. Écrivez l'implémentation des instructions de déplacement `>` et `<` à l'aide des *getters* sur `precedente` et `suivante`.
8. Dans la classe `Memoire`, écrivez deux méthodes `incremente` et `decremente` qui vous permettront d'implémenter les instructions `+` et `-`.
9. Implémentez l'instruction `.` à l'aide du *getter* sur `valeur`.
10. Pour implémenter l'instruction `,`, utilisez un `Scanner` ainsi que sa méthode `nextInt()` pour demander un entier interactivement à l'utilisateur. Vous aurez besoin de la directive `import java.util.Scanner;` et de consulter l'API. Utilisez le *setter* sur `valeur` pour modifier la valeur de la case mémoire courante.
11. Implémentez les instructions `[` et `]`.
    Indication : le but est d'implémenter des *saut* en modifiant la valeur de la variable de votre boucle d'exécution du programme. Pour cela, écrivez des méthodes statiques `fermantCorrespondant(String programme, int depart)` et `ouvrantCorrespondant(String programme, int depart)` permettant de renvoyer la position du délimiteur fermant / ouvrant correspondant. Dans un premier temps, considérez que les délimiteurs de boucle `[` et `]` ne peuvent être imbriquées.
12. Testez votre interprète sur le programme donné en exemple. Testez également sur ce programme : `,>,[-<->]<.` Que calcule-t-il ?
13. Modifiez vos méthodes `fermantCorrespondant` et `ouvrantCorrespondant` de façon à autoriser les boucles imbriquées.
14. Testez sur ce programme : `>>,<[>[<<+<+>>>-]<<[>>+<<-]>-]>[-]<<[-]<.` Que calcule-t-il ?