université PARIS DIDEROT PARIS 7

CONCEPTS INFORMATIQUES CI2

Feuille 01 : Piles & Applications

Comme en amphi, nous utilisons la classe `Stack` et certaines de ses variantes comme `Stack<Integer>`, `Stack<String>`, `Stack<String[]>`, etc.

1 (Partiel 2009) Pour chaque question, commencer par faire un ou des schémas avant d'écrire le code.

1. Écrire une fonction de prototype `void afficher(Stack<Integer>)` qui affiche verticalement, fond de pile vers le bas, le contenu de la pile passée en paramètre ; celle-ci doit être intacte après l'exécution.
2. Écrire une fonction de prototype `void transvaser(Stack<Integer>, Stack<Integer>)` qui déplace le contenu de la première pile vers la deuxième de sorte que l'ordre soit inversé.
3. Écrire une fonction de prototype `void deplacer(Stack<Integer>, Stack<Integer>)` qui déplace le contenu de la première pile vers la deuxième de sorte que l'ordre soit maintenu.
4. Écrire une fonction de prototype `void deplacerPairImpair(Stack<Integer>, Stack<Integer>)` qui déplace le contenu de la première pile vers la deuxième de sorte que tous les nombres pairs soient au-dessous des nombres impairs (l'ordre des pairs entre eux et des impairs entre eux n'importe pas ici).
5. Écrire une fonction de prototype `void copierPairs(Stack<Integer>, Stack<Integer>)` qui copie les entiers pairs de la première pile vers la deuxième de sorte que la première pile soit intacte (après l'exécution) et que l'ordre des pairs entre eux soit le même dans les deux piles.

2 (Partiel 2011) On considère le problème des tours de Hanoi : déplacer n disques de diamètres différents d'une tour de départ 1 à une tour d'arrivée 2 en passant par une tour intermédiaire 3 et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide (on suppose que cette règle est également respectée dans la configuration initiale).

Expliquer, le plus précisément possible, l'exécution (pour $n = 3$) de la solution suivante :

```
import java.util.*;
class HanoiPile{
    static void hanoi(int n){
        Stack<Integer[]> p=new Stack<Integer[]>();
        Integer[] a={1,2,3,0};      a[3]=n;
        p.push(a);
        while(!p.empty()){
            Integer[] e=p.pop();
            if (e.length == 2 || e.length == 4 && e[3] == 1)
                System.out.println(e[0] + "-->" + e[1]);
            else{
                a=new Integer[4]; a[0]=e[2]; a[1]=e[1]; a[2]=e[0]; a[3]=e[3]-1;
                p.push(a);
                a=new Integer[2]; a[0]=e[0]; a[1]=e[1];
                p.push(a);
                a=new Integer[4]; a[0]=e[0]; a[1]=e[2]; a[2]=e[1]; a[3]=e[3]-1;
                p.push(a);
            }
        }
    }
    public static void main(String[] args){
        hanoi(3);
    }
}
```

3 (Partiel 2017) Considérons l'expression

```
( 5 / ( 3 - 1 ) ) * ( 2 + 4 ).
```

1. Dessiner son arbre syntaxique.
2. Donner sa forme préfixe
3. Donner sa forme postfixe.
4. Décrire l'évolution du contenu de la pile (chaque `push` et chaque `pop`) lors de
   (a) l'évaluation de sa forme postfixe ;
   (b) l'évaluation de sa forme préfixe ;
   (c) la transformation de sa forme infixe en sa forme postfixe.

4 Écrire toutes les fonctions de conversion entre les formes infixe, préfixe, et postfixe.

5 Cet exercice s'intéresse aux expressions bien parenthésées.

1. Écrire une fonction de prototype `boolean bienParenthesee(String)` qui vérifie si une chaîne de caractères est bien parenthésée en utilisant une pile. En particulier, elle ignore tous les caractères autres que « `(` » et « `)` » (non pertinents). La fonction :
   – renvoie `true` pour des mots tels que `()`, `()(())` ou `(((())()))`
   – renvoie `false` pour des mots tels que `)(`, `(()(` ou `(((()))))`.
   Pourrait-on facilement se passer d'une pile ?
2. Modifier la fonction pour qu'elle puisse reconnaître les expressions bien parenthésées utilisant deux types de parenthèses, `( )` et `[ ]`. Pour cette question, la fonction :
   – renvoie `true` pour des mots tels que `()[]`, `([()])` ou `[(()())]([[]()])`
   – renvoie `false` pour des mots tels que `([)]`, `()[` ou `[()[]]]`.
   Pourrait-on facilement se passer d'une pile ?

6 (Partiel 2014) Un document XML est un document texte structuré tel le document suivant :

```
<concepts>
        <enseignants>
                <enseignant>Machin</enseignant>
                <enseignant>Bidule</enseignant>
        </enseignants>
        <etudiants>
                <etudiant>Eleanor</etudiant>
                <etudiant>Fares</etudiant>
                <etudiant>Thobias</etudiant>
        </etudiants>
</concepts>
```

Nous nous intéressons dans cet exercice aux *balises*, c'est-à-dire aux éléments structurants du document, comme `<etudiant>` ou `</enseignant>`. Ces balises viennent par couples : à toute balise ouvrante `<balise>` correspond nécessairement une balise fermante `</balise>` plus loin dans le document. D'autre part, entre deux balises ouvrante/fermante associées, on peut imbriquer autant de couples ouvrant/fermant que l'on veut, du type que l'on veut ; mais il est interdit d'entremêler deux couples. L'autre particularité est qu'un document XML est toujours constitué d'un couple de balises englobant tout le reste (le couple `<concepts></concepts>` dans l'exemple).
On suppose qu'un document XML est encodé dans une classe `DocumentXML` et on dispose d'une méthode de prototype `String getNextTag()` qui, appelée itérativement, renvoie dans l'ordre les balises XML d'un document (le reste du document est proprement ignoré). Lorsqu'il n'y a plus de balises à lire, la référence renvoyée est simplement `null`. Par exemple, si la variable `doc` de type `DocumentXML` contient une référence vers le document XML présenté ci-dessus, le premier appel `doc.getNextTag()` renverra la chaîne de caractères `"<concepts>"` et l'appel suivant `doc.getNextTag()` renverra la chaîne `"<enseignants>"`.

1. Donner trois exemples de documents mal formés du point de vue de la structure XML. Attention, les malformations doivent être de nature différente.
2. Écrire une méthode de prototype `boolean estCorrect(DocumentXML)` qui teste si le document lu via des appels à `getNextTag` est un document XML correctement formé.