Solutions

Exercice 1

Que peut-on dire sur l'exécution de tryStack1 et tryStack2 ?

D'un point de vue exécution, aucune différence n'est à noter. Les 2 algorithmes ont l'air d'avoir le fonctionnement suivants :

```
* Création d'une liste de 50 entiers pairs, ordonnés de façon croissante.
* Inversion de l'ordre de la liste (valeurs décroissantes)
```

Décrire le déroulement de l'appel à tryStack1 et tryStack2.

Décortiquons les maisn de tryStack1 et tryStack2 (qui sont identique). Nous avons d'abord la création d'un tableau d'entiers de taille 50, nommé *arr*. A chaque élément i du tableau, on associe la valeur 2*i

```
Integer[] arr = new Integer[50];

for(int i=0; i<50; i++){
   arr[i]=new Integer(i*2);
}
```

La suite du programme va imprimer le tableau arr, à l'aide la function **printA**, puis inverser le tableau avec la fonction **reverse** et imprimer de nouveau le tableau.

```
printA(arr);
arr = reverse(arr);
printA(arr);
```

Cela correspond bien à ce que nous avons pu observer lors de l'exécution.

Quels sont les différences d'implémentation entre tryStack1 et tryStack2 ?

La seule différence est dans la fonction **reverse**. Regardons son fonctionnement afin de comprendre cette différence.

La classe utilisée pour créer la variable S est différente. Leur différence d'implémentation est sur le choix de la structure de donnée utilisées pour gérer la pile. **ArrayStack** choisit de gérer la pile avec un tableau, **NodeStack** fait le choix d'une implémentation dynamique avec la classe **Node**. On remarquera que **Node** à une structure de liste chaînée (simple).

```
// Pour tryStack1
ArrayStack S = new ArrayStack(a.length);

// Pour tryStack2
NodeStack S = new NodeStack();
}
```

Le reste de la fonction va d'abord créer un tableau **b**. On va ensuite pusher tous les éléments de a (le tableau en paramètre de la fonction) dans la pile **S** puis récupérer les éléments de **S** et les stocker dans **b**. Le fonctionnement d'une pile fait que l'enchaînement push/pop d'un tableau entier inverse sont ordre.

```
Integer[] b = new Integer[a.length];

for (int i=0; i < a.length; i++){
   S.push(a[i]);
}

for (int i=0; i < a.length; i++){
   b[i] = (Integer) (S.pop());
}
```

Qu'a t'on voulu mettre en avant ?

2 éléments sont mis en avant ici :

- l'intérêt d'un langage orienté objet tel que Java. Celui ci permet, en une ligne de code, de changer rapidement d'implémentation d'une liste, sans modifier le reste du programme
- comment mettre en place de différente façon une structure de pile.

A noter que Java possède sa propre gestion de piles, nommée **Stack**.

Exercice 2

Nous allons fournir ici quelques explications. La solution se trouve dans le fichier **DLinkedList.java**

InsertNode(ListNode nNode, ListNode pAfter)

On veut insérer un noeud (nNode) à la suite d'un autre (pAfter). Cela implique 4 changement sur les pointeurs associés aux noeuds. Pour simplifier, nous nommerons pNext le noeud pointé par pAfter.

Nous devons faire d'abord les changements suivants :

- le noeud suivant pAfter doit devenir celui de nNode
- le noeud précédent nNode devient pAfter

```
nNode.next = pAfter.next;
nNode.previous = pAfter;
```

Le pNext doit changer de précédent. Cela n'est possible que si pNext n'est pas **null**. S'il est nul, alors nNode devient le nouveau noeud de fin de liste.

```
if (pAfter.next != null) {
   pAfter.next.previous = nNode;
} else {
   lastNode = nNode;
}
```

C'est une fois ces changements effectués que nous pouvons attribuer nNode comme nouveau suivant de pAfter.

```
pAfter.next = nNode;
```

RemoveNode(ListNode nNode)

On veut supprimer un noeud d'une liste doublement chaînée. Nous avons alors 2 cas particulier à gérer :

- le début de liste

```
if (nNode.previous == null) {
  firstNode = nNode.next;
} else {
  nNode.previous.next = nNode.next;
}
```

- la fin de liste

```
if (nNode.next == null) {
  lastNode = nNode.previous;
} else {
  nNode.next.previous = nNode.previous;
}
```

Exercice 3

Le principe de base est le suivant :

- on crée une pile, qui stockera toutes les parenthèses ouvrantes (PO) - i.e. ([{
- on parcourt ensuite l'ensemble des éléments de la chaine de parenthèses
  - si l'on trouve une parenthèse ouvrante, on la stocke dans la pile (push)
  - dès que l'on trouve une parenthèse fermante (PF) - i.e.)]} - nous avons les choix suivants
  - soit la pile est vide, on a donc trop de PF par rapport aux PO, c'est donc une erreur
  - sinon, on dépile (pop) le premier élément. S'il correspond à la PF, c'est ok. Sinon il y a erreur.
- si la pile n'est pas vide à la fin, nous avons trop de PO par rapport aux PF, c'est donc une erreur.