Sokoban : étape 2

Aujourd'hui, et durant quelque semaines, nous allons nous atteler à la réalisation de structures de données, en particulier de séquences qui nous serviront de listes, piles, files et FAP. Par la suite, nous les utiliserons pour réaliser certaines parties du jeu (*pattern* observateur, animations, IA, ...). Dans un premier temps, cela va nous permettre de nous familiariser avec Java à l'aide d'algorithmes et de structures de données déjà connues.

Aujourd'hui, nous allons implémenter une structure de données permettant de stocker une séquence valeurs entières. Les valeurs entières de notre séquence ne sont pas forcément uniques, la même valeur peut se retrouver plusieurs fois dans une séquence. Par contre, elle sont totalement ordonnées par leur position dans la séquence. Nos séquences devront disposer des méthodes suivantes :

- `void insereTete(int element)` :
  insère l'élément nommé `element` en début de séquence (en première position) ;
- `void insereQueue(int element)` :
  insère l'élément nommé `element` en fin de séquence (en dernière position) ;
- `int extraitTete()` :
  extrait et renvoie la valeur de l'élément situé en début de séquence (en première position) ;
- `boolean estVide()` :
  renvoie vrai si et seulement si la séquence est vide.

L'extraction d'un élément est susceptible de lever une exception si la séquence associée est vide. Dans un premier temps utilisez une `RuntimeException` que vous pouvez lever de la manière suivante :

```
throw new RuntimeException("Séquence vide");
```

1 - Représentation sous forme de liste chaînée

Nous commencerons par implémenter nos séquences par une structure de liste chaînée. Écrivez une classe, nommée `SequenceListe`, contenant cette solution. Écrivez également un programme permettant de tester le bon fonctionnement de votre solution. Si vous voulez disposer d'un affichage textuel simple de la séquence, vous pouvez définir une méthode publique :
`public String toString()`
renvoyant sa représentation textuelle, la séquence est alors affichable avec `print` ou `println` de `PrintStream`.

2 - Représentation sous forme de tableau

Malheureusement, le parcours d'une liste chainée est relativement inefficace : les éléments contigus ne sont pas adjacents en mémoire et, en raison du chaînage, l'espace qu'elle occupe n'est pas compact. Nous allons donc implémenter une autre version de la séquence en utilisant un tableau pour stocker les éléments. Avec cette version, le tableau peut être plein lors d'une insertion. Dans un premier temps, gérez ce cas en levant une exception. Ecrivez la classe `SequenceTableau` associée et un programme de test pour tester cette nouvelle implémentation.

3 - Améliorations

Pour rester efficace lors des insertions et des suppressions, nous souhaitons éviter de décaler les éléments en mémoire. Pour cela, il suffit de considérer le tableau comme un tampon circulaire. De plus, afin d'éviter de tomber sur une erreur lorsque la séquence est pleine, nous pouvons redimensionner le tableau pour lui permettre de grossir dynamiquement. Implémentez ces deux améliorations et testez les. Pour le redimensionnement, une bonne stratégie est de doubler la taille du tableau lorsque celle-ci est insuffisante, expliquez pourquoi.

4 - Exceptions

Que se passe-t-il si les exceptions que vous levez dans les séquences sont de type `Exception` et non `RuntimeException` ? D'après vous que faudrait-il faire dans ce cas ? Après avoir répondu à ces questions, revenez au type `RuntimeException`. D'après vous, pourquoi le type `RuntimeException` est-il approprié ici ?