TP n°9

Listes chaînées et Piles

On rappelle qu'une `liste` est un modèle de données qui permet de représenter et de manipuler une séquence d'objets contenant chacun une valeur d'un type donné. Les opérations basiques que l'on peut définir sur une telle liste sont :
– $estVide : L \rightarrow Bool$ indique que la liste est vide (elle ne contient pas d'éléments)
– $premier : L \rightarrow V$ donne la valeur de la tête de liste
– $suivant : L \rightarrow L$ donne la suite de la liste
– $ajouteEnTete : L \times V \rightarrow L$ ajoute une valeur en tête de liste

Exercice 1 *Listes chaînées*

La classe `Element` *va représenter un* maillon de la chaîne *:*

```
import fr.jussieu.script.Deug;

public class  Element {
    // propriétés
    private int valeur;
    private Element suivant;  // élément suivant dans la liste
}
```

1. *Définir le constructeur par défaut qui initialisera la valeur de l'élément à* 0.
2. *Définir un second constructeur qui prend en argument un entier destiné à l'initialisation de la valeur de l'élément.*
3. *Ecrire les modifieurs et accesseurs nécessaires.*

La classe Liste

```
import fr.jussieu.script.Deug;
public class  Liste {
    // propriétés
    private int     nombreElements; // nombre d'éléments de la liste
    private Element tete;           // l'élément en tête de liste
    private Element queue;          // l'élément en fin de liste

    // méthodes publiques
    public boolean estVide() {
       return (tete == null);
    }
    public int longueur() {
```

```
        return nombreElements;
    }
}
```

1. *Définir le constructeur par défaut.*
2. *Écrire une méthode publique* `vider` *qui vide la liste.*
3. *Ecrire une methode publique* `ajoutEnTete(Element e)`*.*
4. *Ecrire une methode publique* `ajoutEnQueue(Element e)`*.*
5. *Écrire une méthode publique* `recherche` *qui retourne l'élément contenant un entier passé en paramètre (*`null` *le cas échéant).*
6. *Écrire la méthode publique* `toString` *qui retourne la chaîne de caractères correspondant au contenu de la liste.*
7. *Écrire une méthode publique* `Supprimer` *qui supprime un élément donné de la liste.*
8. *Écrire une méthode publique* `inverse` *qui inverse le contenu de la liste. Si par exemple on inverse la liste* $L = [1, 2, 3]$ *alors* $L = [3, 2, 1]$.
9. *Ecrire la méthode publique* `insereApres(int x, Element y)` *qui insere l'element y derrière l'element de valeur x. l'ajout ce fait en fin si x n'existe pas.*
10. *Ecrire la méthode publique* `insereApres(Element x, Element y)` *qui insere l'element y derrière l'element x. l'ajout ce fait en fin si x n'existe pas.*
11. *Définir une méthode publique* `precedent` *qui prend en argument un élément et retourne l'élément précédent dans la liste (*`null` *le cas échéant).*

Exercice 2 *introduction aux Piles*

Une pile est une structure de données qui fonctionne comme une pile d'assiettes, l'assiette qui est au-dessus étant le sommet de la pile. Avec une structure de données de type pile, les seules opérations possibles sont empiler, dépiler *et* estVide. *Lorsque l'on dépile, on commence par l'élément qui se trouve sommet. Une pile possède une structure LIFO (Last In, First Out) en opposition aux files qui ont une structure FIFO (First In, Fist Out).*

On se restreint d'abord à des piles d'entiers dont la taille est bornée par `tailleMax`*. Lorsque la pile est pleine on ne peut plus empiler et lorsque la pile est vide on ne peut pas dépiler, en pratique on teste le vide avant de dépiler. On peut implanter ces piles avec la classe suivante :*

```
class Pile{
    private int tailleMax; // taille maximale de la pile
    private int[] contenu; // éléments contenus dans la pile
    private int sommet; // position du sommet de la pile dans le tableau
}
```

L'élément `contenu[sommet]` *est l'élément qui se trouve en haut de la pile. Par convention, la pile est vide si* `sommet` *vaut -1.*

1. *Ecrire un constructeur* `Pile(int taille)` *qui initialise une pile avec une taille maximale donnée.*
2. *Ecrire les méthodes* `empiler`*,* `depiler` *et* `estVide`*.*