Algorithmique avancée
TD n° 2
Implémentation des SDA

I : Implémentation des listes chaînées :
On donne la spécification en Java des opérations sur les listes, dans l'interface ListeSimple ci-dessous. Les données sont du type Object, ce qui exclut les données primitives.

```
public interface ListeSimple {
public void rembobiner() //place le curseur devant la liste
public Object voirSuivant()  /* renvoie l'élément qui suit celui sous le curseur (le premier s'il
        était devant la liste). Quand il n'y a pas de suivant, renvoie null */
public void avancer()  /* place le curseur sur l'élément renvoyé par voirSuivant */
public void ajouter(Object don) /* insère don en position de suivant par rapport au curseur
        (en tête si le curseur est devant la liste) */
public void supprimer() /* Supprime la donnée en position de suivant par rapport au curseur.
        Supprimer quand le curseur est en fin de liste est une erreur */
public boolean estEnFin() /* vrai quand il n'y a pas de suivant */
}
```

On rappelle la classe Cellule étudiée au TD 12 de BDP 1ère année :

```
public class Cellule
 { private Object elem;
   private Cellule suivant;

   public Cellule ( Object o,Cellule suivant )     {
       elem = o;  this.suivant = suivant ;
     }
  public Cellule ( Object o)  { this(o,null) ;  }

   public Object getElem() { return(elem); }
   public Cellule getSuivant() { return(suivant); }
   void setElem(Object o) { elem = o;}
   void setSuivant(Cellule s) { suivant = s;}
   public  String toString() { return (elem.toString());}
} // end class Cellule
```

On donne aussi le squelette d'une classe ListeCellule implémentée avec des **sentinelles**. Une sentinelle est une cellule sans donnée, qui sert seulement à pouvoir écrire les algorithmes plus commodément.

```
public class ListeCelluleSimple{
    Cellule debut, fin ; //les sentinelles
    Cellule curseur ;

    /** crée une liste vide et rembobinée*/
   public ListeCelluleSimple() {
```

```
        fin = new Cellule(null) ;
        fin.setSuivant(fin) ;
        debut=new Cellule(null, fin) ;
        curseur = debut ;
        } // fin du constructeur

          /* A compléter */

}//fin de ListeCelluleSimple
```

1. Implémenter les méthodes de l'interface ListeSimple dans la classe ListeCelluleSimple
2. En première année, vous avez utilisé l'interface Liste suivante :

```
public interface Liste
  {  public void rajouterEnTete(Object item);
     public void rajouterEnQueue(Object item);
     public Object retirerEnTete() throws Exception;
     public Object retirerEnQueue() throws Exception;
     public Object voirTete() throws Exception;
     public Object voirQueue() throws Exception;
     public boolean estVide();
  }
```

   Ré-écrivez les méthodes de l'interface Liste avec celles de l'interface ListeSimple. L'inverse est-il possible ?

3. Dans une ListeCelluleSimple de 1000 éléments, combien d'opérations demande la méthode rajouterEnQueue() ? On peut ajouter une variable membre de façon que cette méthode se fasse en deux opérations ; trouvez cette modification et écrivez la nouvelle méthode rajouterEnQueue().
   Implantez une file d'attente dans une liste chaînée.

II. Piles

1. On rappelle l'interface Pile déjà vue l'an dernier :

```
public interface Pile   {
      public void empiler(Object item);
      public Object depiler() throws  Exception;
      public Object regarderSommet() throws  Exception;
      public boolean estVide();
  }  // end interface Pile
```

   Donnez une implémentation d'une pile d'objets à l'aide des méthodes de l'interface ListeSimple. Écrivez la classe PileObjet qui correspond en sous-classant ListeCelluleSimple. L'interface Pile peut-elle spécifier une pile dont les données sont des entiers (on dit que l'interface spécifie la classe quand on peut déclarer que la classe implémente cette interface sans erreur de compilation) ?

2. Implémentation d'une pile dynamique d'entiers en java.
   Vous avez à la fin de la question un squelette de la classe PileInt. Suivez le plan de travail ci dessous :

- écrivez une version provisoire de la classe PileInt qui implémente toutes les méthodes sans test d'erreur et testez-la (testPile.java est fourni)
- ajoutez les tests d'erreur 'pile vide' et les exceptions nécessaires
- marquez les endroits où un test d'erreur 'pile pleine' serait nécessaire (vous pouvez ne pas écrire le traitement ou le simplifier)
- remplacez le traitement de l'erreur 'pile pleine' par la technique suivante :
    i. demander un tableau dont la taille est le double de la taille actuelle (qui est dans la v.m. taille)
    ii. recopier le tableau actuel dans ce nouveau tableau
    iii. remplacer le tableau actuel par le nouveau
    iv. continuer l'opération en cours, maintenant il n'y a plus de débordement.

```
// ------------------------------------------------------------------
// Fichier.........: Pileint.Java
// Auteur.......: F. Levy
// Cree Le......: Wednesday 20 October 2004, 16 H 23.
//
// Description..: Pile d'entiers implémentée à l'aide d'un tableau
// dynamique.
// *** Histoire des revisions ***
//
// ------------------------------------------------------------------

public  class PileInt{
    static final int taillebase = 64 ; /* variable de classe non
                                          modifiable */
    protected int stock[] ;
    protected int taille, sommet ;

    public  PileInt() {
        taille = taillebase ;
        stock = new int[taille] ;
        sommet = 0 ; // sommet donne la prochaine place libre
        } // fin du constructeur

   /* à compléter */

} // Fin de PileInt
```