TD Algorithmique - Feuille 4

Listes

On calculera la complexité de chacune des méthodes.

1. **Classe Liste**

   Pour chaque exercice, écrire les méthodes dans la classe Liste (cas de la liste vide, appel sur la tête), et dans la classe Chainon (cas du dernier chainon, appel récursif sur le reste).

   - Ecrire la méthode "public String toStringIter()" qui renvoie une chaine de caractères décrivant tous les éléments de la liste de façon itérative.
   - Ecrire la méthode "public Object chercheElement(int i)" qui renvoie le i^{eme} élément de la liste (on suppose que l'on numérote à partir de 0).
   - Ecrire la méthode "public void supprimerElement(int i)" qui supprime le i^{eme} élément de la liste.
   - Ecrire la méthode "public Liste intersection(Liste l)" qui renvoie une liste contenant les éléments qui appartiennent à la fois à *this* et à l. Par exemple, si l1 = {"un", "deux", "trois"} et l2 = { "trois", "quatre", "un", "huit", "cinq"} alors l1.intersection(l2) doit contenir "un" et "trois".

2. **Listes ordonnées**

   - Ecrire la méthode "public ObjetAClef cherche(int c, int i)" qui renvoie le i^{eme} élément de clef c le plus à gauche.
   - Ecrire la méthode récursive "public ListeOrdonnee fusionneRecursif(ListeOrdonnee l)" telle que l1.fusionneRecursif(l2) renvoie la liste qui contient les éléments de $l1$ et ceux de $l2$ triés par ordre croissant.

3. **Ensembles**

   On peut représenter les ensembles par des listes ordonnées. Dans le cas d'un ensemble les objets sont des entiers et la liste ordonnée ne peut pas contenir deux fois le même élément.

   - Implémenter les opérations élémentaires *appartient*, *union* et *estInclus* sur les ensembles dans la classe "Ensemble" donnée en annexe. Commenter chaque méthode en notant les antécédents et conséquents, et la complexité dans le pire des cas.

     Exemples :

     appartenance : 5 appartient à $\{3, 5, 8, 17\}$, 4 n'appartient pas à $\{3, 5, 8, 17\}$

     union : $\{3, 5, 8, 17\}$ union $\{2, 3, 4, 5, 9, 17, 19\}$ vaut $\{2, 3, 4, 5, 8, 9, 17, 19\}$

     inclusion : $\{3, 4, 9, 19\}$ est inclus dans $\{2, 3, 4, 5, 9, 17, 19\}$
   - Expliquer pourquoi il est intéressant d'utiliser des listes ordonnées plutôt qu'un tableau ou un vecteur pour coder les ensembles

Annexes

```
-----------------------------------------------
CLASSES ChainonEnsemble et Ensemble
-----------------------------------------------
public class ChainonEnsemble {
    protected int element; // l'élément
    protected ChainonEnsemble reste; // le lien vers le chainage suivant
    /**
     * Construit un ensemble à un élément.
     */
    public ChainonEnsemble(int x) {
       reste  = null;
       element = x;
    }
    /** Constructeur privé pour les méthodes récursives qui construisent des ChainonEnsembles (union, intersection) */
    protected ChainonEnsemble(int x,ChainonEnsemble r) {
        reste = r;
        element = x;
    }
   /**
     * Teste si le chainon est le dernier
     */
    public boolean dernier() {
        return (reste == null);
    }

  /**
     * Accroche un nouveau Maillon contenant l'objet x <br>
     * antécédent : la liste est triée par ordre croissant et ne contient deux fois la même valeur <br>
     * conséquent : la liste contient x, est triée par ordre croissant et ne contient deux fois la même valeur <br>
     * complexite : O(n)<br>
     */
    public void ajouter(int x) {
        if (dernier())
            reste = new ChainonEnsemble(x);
        else if (reste.element > x)
            reste = new ChainonEnsemble(x,reste);
        else if (x > reste.element) reste.ajouter(x);
        // si x est déjà dans la liste on ne fait rien
    }
    /**
     * antécédent : c est un entier
     * conséquent : true s'il y a un élément de clef c dans l'ensemble this
     * complexité :
     */
    public boolean appartient(int c) {
       // A COMPLETER
    }
    /**
     * antécédent : l est un ensemble <br>
     * conséquent : true si tous les éléments de this sont aussi dans l
     * complexité : theta(l1 + l2) où l1 et l2 sont les longueurs des deux ensembles
     */
     public boolean estInclus(ChainonEnsemble l) {
       // A COMPLETER
    }
    /**
```

```
     * antécédent : l est un ensemble <br>
     * conséquent : un ensemble  contenant les éléments de l et ceux de this <br>
     * complexité :
     */
    public ChainonEnsemble union(ChainonEnsemble l) {
       // A COMPLETER
    }
    /**
     * antécédent : l est un ensemble <br>
     * conséquent : un ensemble  contenant les éléments qui appartiennent à la fois à l  et à this <br>
     * complexité :
     */
    public ChainonEnsemble intersection(ChainonEnsemble l) {
       // A COMPLETER
    }
    public String toString() {
        if (dernier()) return element + "";
        else return element + " -> " + reste.toString();
    }
}
-----------------------------------------
/** Une classe pour les ensembles <br>
 * Un ensemble est représenté comme une ListeOrdonnee. Il ne peut pas y avoir deux fois le même élément
 */
public class Ensemble {
    private ChainonEnsemble tete; // le lien vers le chainage
    /**
     * Construit un ensemble vide.
     */
    public Ensemble() {
        tete = null;
    }
    /** Constructeur privé pour les méthodes récursives qui construisent des Ensembles (union, intersection) */
    private Ensemble(ChainonEnsemble c) {
        tete = c;
    }
   /**
     * Teste si l'ensemble est vide
     */
    public boolean vide() {
        return (tete == null);
    }
    private void ajouterDebut(int x) {
        if (vide())
            tete = new ChainonEnsemble(x);
        else tete = new ChainonEnsemble(x,tete);
    }
  /**
     * Accroche un nouveau Maillon contenant l'objet x <br>
     * antécédent : la liste est triée par ordre croissant et ne contient deux fois la même valeur <br>
     * conséquent : la liste contient x, est triée par ordre croissant et ne contient deux fois la même valeur <br>
     * complexite : O(n)<br>
     */
    public void ajouter(int x) {
        if (vide()|| x<tete.element)
           ajouterDebut(x);
        else if (x > tete.element)  tete.ajouter(x);
        // si x est déjà dans la liste on ne fait rien
        }
```

```
    /**
     * antécédent : c est un entier
     * conséquent : true s'il y a un élément de clef c dans l'ensemble this
     * complexité :
     */
    public boolean appartient(int c) {
        // A COMPLETER
    }
    /**
     * antécédent : l est un ensemble <br>
     * conséquent : true si tous les éléments de this sont aussi dans l
     * complexité :
     */
     public boolean estInclus(Ensemble l) {
        // A COMPLETER
    }
    /**
     * antécédent : l est un ensemble <br>
     * conséquent : un ensemble  contenant les éléments de l et ceux de this <br>
     * complexité :
     */
    public Ensemble union(Ensemble l) {
        // A COMPLETER
}
    /**
     * antécédent : l est un ensemble <br>
     * conséquent : un ensemble  contenant les éléments qui appartiennent à la fois à l  et à this <br>
     * complexité :
     */
    public Ensemble intersection(Ensemble l) {
        // A COMPLETER
    }
    public String toString() {
        if (vide()) return " VIDE ";
        else return tete.toString();
    }
}
```