TD n°7

Listes chaînees

Dans ce TD, nous manipulerons uniquement des listes d'entiers que nous implementerons par des listes chaînees.
Une liste chaînee est une suite d'elements formes d'un contenu (ici un entier) et de l'adresse (reference) vers l'element suivant.

```
public class  Element {
  int contenu;
  Element suivant;  // element suivant dans la liste

  Element (int x, Element a) {
    contenu = x;
    suivant = a;
  }
}
```

Dans la premiere partie on ne manipulera que des `Element` qui a une structure recursive, dans la seconde partie la liste d'`Element` sera encapsule dans `Liste` a n d'ameliorer certaines operations et pourvoir la manipuler comme un objet.

1 Liste chaînée basique

Les operations primitives sur les listes sont
- `tete` qui renvoie le contenu de l'element en tête de liste.
- queue qui renvoie la queue de la liste (c'est a dire la liste privee de la tête).
- `ajouter` qui renvoie une liste avec un nouvel element place en tête, les autres elements sont partages avec la liste l passee en argument.

```
static int tete(Element l){
  return l.contenu;
}

static Element queue(Element l){
  return l.suivant;
}

static Element ajouter(int x, Element l){
  return new Element(x, l);
}
```

Remarques :
{ ici une liste est identi ee a son premier element,
{ la methode `ajouter` ne change pas la liste passee en argument, elle cree une liste qui contient une element suivi de l'ancienne liste.

Exercice 1 *A quoi correspond la liste vide ?*
Écrire une méthode `boolean estVide(Element l)` *qui teste si une liste est vide.*

Exercice 2 *Soit* `liste` *de type* `Element`*, que font les opérations suivantes :*
– `tete(ajouter(x, liste))`
– `queue(ajouter(x, liste))`
– `ajouter(tete(liste), queue(liste))`

Exercice 3 *Écrire une méthode récursive* `int longueur(Element l)` *qui renvoie la longueur de la liste* `l` *passée en argument.*
Quelle est sa complexité ?

Exercice 4 *Écrire une méthode récursive* `void affiche(Element l)` *qui affiche la liste dans la sortie standard.*

Exercice 5 *Écrire une méthode récursive* `Element copie(Element l)` *qui renvoie une copie de la liste.*

Exercice 6 *Écrire une méthode récursive* `boolean recherche(int x, Element l)` *qui recherche si un entier* `x` *est contenu dans la liste.*

Exercice 7 *Nous avons fait le choix précédement d'écrire les méthodes récursivement, on aurait pu aussi bien les écrire itérativement. Donnez une version itérative de* `boolean recherche(int x, Element l)`

Exercice 8 *Que font les deux méthodes suivantes ? En quoi sont elles différentes ? Expliquer à l'aide d'un dessin sur un exemple simple a=1,2,3 et b=4,5,6.*

```
static Element concat(Element a, Element b){
  if (a == null) return b;
  return ajouter(a.contenu, concat(a.suivant, b);
}

static Element fusion(Element a, Element b){
  if (a == null) return b;
  a.suivant = fusion(a.suivant, b);
  return a;
}
```

Exercice 9 *Si possible, les réécrire en supposant que* `contenu` *et* `suivant` *sont* `private` *et qu'il y ait des accesseurs* `contenu()` *et* `suivant()`*, mais pas de modifieur.*

Exercice 10 *Écrire une méthode* `static Element supprimer(int x, Element l)` *qui renvoie la liste sans la première occurrence de* `x`.

Exercice 11 *Sachant que les listes peuvent partager des élements, supposons que deux listes* `l1` *et* `l2` *se rejoignent sur un élément qui contient une valeur* `x` *qui n'apparait qu'une fois, que se passe t'il pour* `l2` *lorsqu'on effectue* `l1.supprimer(x)` *? Pour palier à ce problème, écrire une nouvelle méthode* `supprimer` *qui ne modifie pas la liste* `l` *passée en argument, mais renvoie une nouvelle liste sans la première occurrence de x telle que les éléments avant x sont une copie des éléments de* `l` *et que les éléments après x soient communs à* `l`.
Quel est l'avantage à faire cela et à utiliser `concat` *plutôt que* `fusion` *?*

Exercice 12 *Écrire une méthode récursive* `Element inverser(Element l)` *qui crée et renvoie une nouvelle liste dans laquelle les entiers sont inversés à l'aide d'une méthode récursive* `inverserAux` *qui prend deux listes* `l1` *et* `l2` *en argument et renvoie une liste qui est la tête de* `l1` *suivie de* `l2`.

2 Liste chaînée encapsulée

On peut accelerer certaines methodes comme `int longueur(Element l)` ou permettre l'ajout rapide en n de liste en sauvegardant de l'information supplementaire, mais aussi rendre la manipulation des listes plus facile pour un utilisateur en l'encapsulant dans une classe `Liste`.

```
public class  Liste {
    private int     nombreElements; // nombre d'elements de la liste
    private Element tete;           // l'element en tête de liste
    private Element fin;          // l'element en fin de liste

    public boolean estVide() {
       return (tete == null);
    }

    public int longueur() {
       return nombreElements;
    }
}
```

Exercice 13 *Écrire le constructeur de la liste vide.*

Exercice 14 *Écrire une méthode publique void vider() qui vide la liste.*

Exercice 15 *Écrire une méthode publique* `void ajouter (int x)` *qui ajoute un élément en tête de la liste à l'aide de la méthode* `ajouter` *écrite dans la classe* `Element`.

Exercice 16 *Écrire une méthode publique* `void ajouterFin(int x)` *qui ajoute un élément en fin de la liste.*
Quelle est la complexité ? Quelle aurait été la complexité sans sauvegarder l'élément en fin de liste ?