TD n°6

Listes chaînées

Dans ce TD, nous manipulerons uniquement des listes d'entiers que nous implémenterons par des listes chaînées. Une liste chaînée est une suite d'éléments formés d'un contenu et d'une référence vers l'élément suivant. Ici le contenu sera entier.

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

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

Dans la première partie on ne manipulera que des objets de la classe `Element` qui a une structure récursive, dans la seconde partie la liste d'`Element` sera encapsulé dans `Liste` afin d'améliorer certaines opérations. Par conséquent, toutes les méthodes dans la première partie sont statiques (sinon, on ne pourrait pas manipuler la liste vide), tandis que toutes les méthodes dans la deuxième partie sont dynamiques.

1 Liste chaînée basique

Les opérations primitives sur les listes sont :
- `tete` qui renvoie le contenu de l'élément en tête de liste.
- `queue` qui renvoie la queue de la liste (c'est à dire la liste privée de la tête).
- `ajouter` qui renvoie une liste avec un nouvel élément placé en tête, les autres éléments sont partagés avec la liste `l` passée 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);
}
```

Remarque : La méthode `ajouter` ne change pas la liste passée en argument, elle crée une liste qui contient un élément suivi de l'ancienne liste.

Exercice 1 À 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. Combien fait elle d'appels récursifs ?

Exercice 4 Écrire une méthode récursive `int dernier(Element l)` qui renvoie la valeur du dernier élément d'une liste.

Exercice 5 Écrire une méthode récursive `void afficher(Element l)` qui affiche la liste sur la sortie standard.

Exercice 6 Écrire une méthode récursive `Element copier(Element l)` qui renvoie une copie de la liste.

Exercice 7 Écrire une méthode récursive `boolean rechercher(int x, Element l)` qui regarde si l'entier `x` est contenu dans la liste.

Exercice 8 Nous avons fait le choix précédemment d'écrire les méthodes récursivement, on aurait pu aussi bien les écrire itérativement.

Donnez une version itérative de `boolean rechercher(int x, Element l)`.

Exercice 9 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 concatener(Element a, Element b){
  if (a == null) return b;
  return ajouter(a.contenu, concatener(a.suivant, b));
}

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

Exercice 10 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 11 Écrire une méthode `Element supprimer(int x, Element l)` qui renvoie la liste `l` sans la première occurrence de `x`.

Exercice 12 Sachant que les listes peuvent partager des éléments, supposons que deux listes `l1` et `l2` se rejoignent que se passe-t-il pour `l2` lorsqu'on effectue `supprimer(x, l1)` si `x` apparaît pour la première fois dans la partie commune à `l1` et `l2` ?

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 `concatener` plutôt que `fusioner` ?

Exercice 13 Écrire une méthode récursive `Element inverser(Element l)` qui crée et renvoie une nouvelle liste dans laquelle l'ordre des entiers est inversé (le début de `l` est à la fin de `inverser(l)` et réciproquement). Pour cela, écrire une méthode récursive `inverserAux` qui prend deux listes `l1` et `l2` en argument, prend l'élément de tête de `l1` pour le mettre devant `l2` et recommence jusqu'à vider `l1`.

2 Liste chaînée encapsulée

On peut accélérer certaines méthodes comme `int longueur(Element l)` ou permettre l'ajout rapide en fin de liste en sauvegardant de l'information supplémentaire en l'encapsulant dans une classe `Liste`.

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

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

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

Exercice 14 Écrire le constructeur de la liste vide.

Exercice 15 Écrire une méthode publique `void ajouterTete (int x)` qui ajoute un élément en tête de la liste.

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 ?