Université Paris-Est Créteil
Licence MIAGE - L3

Programmation orientée objet en Java
Année 2009-2010

Examen

Durée : 3h. Aucun document autorisé.

Les exercices 1, 2 et 3 sont indépendants et peuvent être traités dans n'importe quel ordre.

Exercice 1 : (4 points)

Dans cet exercice, nous nous intéressons à la définition d'une liste chaînée dont les éléments sont des couples (clé, valeur) de type entier (aussi appelés associations). La liste chaînée va être définie en utilisant des cellules chaînées les unes aux autres.

On donne l'interface suivante :

```
interface Cell {
    // Renvoie true si la liste est vide
    public boolean isEmpty();
    // Renvoie la prochaine cellule de la liste
    public Cell getNext() throws Empty;
    // Remplace la prochaine cellule par c
    public void setNext(Cell c) throws Empty;
    // Renvoie la cle de cette cellule
    public int getKey() throws Empty;
    // Renvoie la valeur de cette cellule
    public int getVal() throws Empty;
    // Remplace la valeur de cette cellule par v
    public void setVal(int v) throws Empty;
}
```

1. Définir l'exception vérifiée `Empty` qui servira à signaler que la liste est vide.
2. Définir une classe `EmptyCell` (sous-type de `Cell`) représentant la cellule vide de fin de liste.

   Son constructeur aura le prototype `public EmptyCell()`.
3. Définir une classe `KeyValCell` (sous-type de `Cell`) représentant un maillon de la liste chaînée.

   Son constructeur aura le prototype `public KeyValCell(int k, int v, Cell c)`, où `k` est la clé, `v` est la valeur et `c` est la suite de la liste à partir de la cellule nouvellement créée.

Exercice 2 : (8 points)

Dans cet exercice, nous nous intéressons à la définition d'une liste d'association triée en utilisant la liste chaînée définie par les classes `EmptyCell` et `KeyValCell` de la question précédente. La particularité de cette liste d'association est de maintenir en permanence les cellules triées dans l'ordre croissant des clés.

1. Définir l'exception vérifiée `UnknownKey` qui servira à signaler l'absence d'une clé cherchée dans la liste.

2. Définir une classe `AssociationList` représentant la liste d'association. L'unique champ de cette classe sera un objet de type `Cell` représentant la tête de la liste. Définir un constructeur ne prenant rien en paramètre, puis les méthodes de prototype :
   - `public boolean isEmpty()` : renvoie `true` si la liste est vide et `false` sinon.
   - `public void add(int k, int v)` : si l'une des cellules de la liste a pour clé `k` alors sa valeur est mise à `v`, sinon une nouvelle cellule contenant l'association `(k, v)` est ajoutée à la bonne position dans la liste.
   - `public int getVal(int k)` : renvoie la valeur associée à la clé `k`, ou lève l'exception `UnknownKey` si la clé `k` n'est pas dans la liste.

Exercice 3 : (2 points)

Dans cet exercice, nous utilisons la liste d'association pour représenter un tableau creux, c'est-à-dire dont seules les cases ayant été affectées ont réellement une valeur. Les indices du tableau seront représentés par les clés de la liste d'association, et les valeurs des cases par les valeurs associées aux clés.

Ainsi la liste d'association contenant les associations `(1, 6)` → `(12, 2)` → `(137, 14)` représentera un tableau creux de `137` cases dont la case `1` contient la valeur `6`, la case `12` la valeur `2` et la case `137` la valeur `14`.

Définir une classe `HollowArray` représentant un tableau creux, avec les méthodes de prototype :

- `public void set(int index, int val)` : affecte la valeur `val` à la "case" `index`.
- `public int get(int index)` : renvoie la valeur présente dans la "case" `index` ou propage l'exception `UnknownKey` si la case n'existe pas.

Exercice 4 : (2 points)

On souhaite rendre toutes nos structures paramétrées par le type des valeurs stockées.

1. Définir l'interface `Cell` générique.
2. Rendre les classes `EmptyCell`, `KeyValCell`, `AssociationList` et `HollowArray` génériques. (modifier le code avec une couleur différente.)

Exercice 5 : (4 points)

On rappelle la définition des interfaces `Iterable<E>` et `Iterator<E>`.

```
interface Iterable<E> {
    public Iterator<E> iterator();
}

interface Iterator<E> {
    public boolean hasNext();
    public E next();
}
```

1. Faire implanter par la classe `AssociationList` l'interface `Iterable` afin qu'elle propose un itérateur sur la liste (dans l'ordre des clés).
2. En utilisant cet itérateur, faire implanter par la classe `HollowArray` l'interface `Iterable` afin qu'elle propose un itérateur (dans l'ordre des cases).