Université Grenoble Alpes
Master MIASHS
Documents autorisés : 1 copie manuscrite recto/verso (non photocopiée)

Programmation 2
Code de Huffman

Le code de Huffman est un code permettant de comprimer des données : l'idée de base est de coder chaque élément d'une liste avec un code de longueur variable, les éléments les plus fréquents sont codés avec un code plus court que les éléments les moins fréquents.
Le code de Huffman est calculé en construisant un arbre de la façon suivante :

1. les éléments, munis de leurs nombres d'occurrences (les noeuds) sont rangés dans une liste ordonnée par ordre croissant de leur nombre d'occurrences.
2. tant que la liste contient au moins deux noeuds, il faut faire :
    1. retirer les deux noeuds x et y de plus faible nombre d'occurrences.
    2. ajouter dans la liste un nœud binaire dont le nombre d'occurrences est la somme des nombres d'occurrences de x et y, et qui possède x en sous arbre gauche et y en sous arbre droit.

Le but de ce TP est de réaliser le codage de Huffman d'une liste d'objets. Les objets les plus fréquents dans la liste seront codés avec un nombre de bits inférieur à ceux qui sont présents peu de fois.

Pour réaliser l'algorithme précédent nous définissons les classes `ListeTriee`, puis `NoeudAbstrait`, `NoeudBinaire` et `Feuille`. Nous définirons ensuite la classe `CodageHuffman` qui permet de sérialiser/désérialiser le codage d'Huffman d'une liste d'objets.

1- Classe ListeTriee (2 points)

Définir la classe `ListeTriee` comme une sous-classe de `LinkedList`. La classe `ListeTriee` maintient une liste ordonnée par ordre croissant de ses éléments. Les éléments ajoutés doivent implémenter l'interface `Comparable<E>`.

```
public class ListeTriee<E extends Comparable<E>> extends LinkedList<E> {
        public boolean add(E n) {
                ...
        }
}
```

2- Classe NoeudAbstrait (2 points)

Ecrire les définitions du constructeur et des méthodes de la classe `NoeudAbstrait`. La méthode `compareTo(E o)` retourne un entier négatif si le nombre d'occurrences de `this` est plus petit que le nombre d'occurrences de `o`, un entier positif nombre d'occurrences de `this` est plus grande que le nombre d'occurrences de `o` et 0 sinon.

```
public abstract class NoeudAbstrait<E> implements Comparable<NoeudAbstrait<E>>, Serializable {
        private int nbOcc;

        public NoeudAbstrait(int nbOcc) {...}

        public int getNbOcc() {...}
```

```
        public int compareTo(NoeudAbstrait<E> o) {...}

        protected abstract void getDictionnaire(Map<E,String> dict, StringBuffer buf);

        public Map<E,String> getDictionnaire() {
                Map<E,String> dict=new HashMap<E,String>();
                getDictionnaire(dict,new StringBuffer());
                return dict;
        }
        public abstract E decode(Iterator<Boolean> elemsCodes);
}
```

3- Classes NoeudBinaires et Feuille (8 points)

Ecrire les définitions des constructeurs et des méthodes `getDictionnaire` et `decode` des classes `NoeudBinaire` et `Feuille`.

La méthode `getDictionnaire` permet de construire récursivement le dictionnaire de traduction. Le paramètre `dict` est un tableau associatif permettant d'associer à chaque symbole son code binaire de Huffman (sous forme de chaine de caractères). Le paramètre `buf` permet de stocker le code binaire en cours de construction. Lorsque l'on descend à gauche un `'0'` est ajouté, lorsque l'on descend à droite un `'1'` est ajouté. Lorsque l'appel arrive sur une feuille le contenu du buffer sera rajouté au dictionnaire.

La méthode `decode` (appliquée à la racine de l'arbre) permet de décoder le code Huffman d'un élément (à partir d'une suite de bits accessible via l'itérateur passé en paramètre).

```
public class NoeudBinaire<E> extends NoeudAbstrait<E>{
        private NoeudAbstrait<E> gauche, droit;

        public NoeudBinaire(NoeudAbstrait<E> g, NoeudAbstrait<E> d) {...}

        protected void getDictionnaire(Map<E,String> dict, StringBuffer buf) {…}

        public E decode(Iterator<Boolean> elemsCodes) {...}
}

public class Feuille<E> extends NoeudAbstrait<E> {
        private E element;

        public Feuille(E elem, int nbOcc) {...}

        protected void getDictionnaire(Map<E, String> dict, StringBuffer buf) {…}

        public E decode(Iterator<Boolean> elemsCodes) {...}
}
```

4- Classe CodageHuffman (8 points)

La classe `CodageHuffman` permet de construire le codage d'Huffman d'une liste d'objets (passée en paramètre du constructeur). La sérialisation d'une instance de cette classe permettra de coder (`writeObject`), ou de décoder (`readObject`) la liste.

1 – Réaliser la méthode `compteOccurrences` qui compte les occurrences de chaque élément de la liste et renvoie un tableau associatif contenant ces nombres d'occurrence (2 points)

2 – Réaliser la méthode `buildTree` qui permet de construire l'arbre de Huffman et de placer sa racine dans l'attribut `tree` (2 points).

3 – réaliser la méthode `writeObject` qui permet de sérialiser l'instance du codage de Huffman de la liste (i.e. l'arbre, le nombre d'éléments de la liste , puis le codage de Huffman de chacun des éléments). (2 points)

4 – réaliser la méthode `readObject` qui permet de lire une instance du codage de Huffman d'une liste qui a été sérialisée via méthode `writeObject`. Pour cela, il faut utiliser `HuffmanIterator` qui permet de lire bit à bit un flux d'entrée (`InputStream`). (2 points)

```
public class CodageHuffman<E> implements Serializable {

	private List<E> elems;

	private NoeudAbstrait<E> tree;

	private CodageHuffman() {}

	public CodageHuffman(List<E> list) {
		elems = list;
		Map<E, Integer> occ = compteOccurrences();
		buildTree(occ);
	}

	private Map<E, Integer> compteOccurrences() {...}

	private void buildTree(Map<E, Integer> occ) {...}

	public List<E> getDecodedElements() {
		return elems;
	}

	private void writeObject(ObjectOutputStream out) throws IOException {
		...
		// utiliser la méthode ecrireChaineBinaire pour ecrire un element encodé
		// nécessaire pour vider le buffer à la fin
		videEtPadding(out);

	}

	private transient StringBuffer buffer;
	public void ecrireChaineBinaire(String s, ObjectOutputStream out) throws IOException {
		if (! s.matches("^[0-1]*$")) new IOException("La chaine est invalide");
		if (buffer==null) buffer=new StringBuffer();
		buffer.append(s);
		while (buffer.length()>7) {
			int b = Integer.parseInt(buffer.substring(0, 8), 2);
			out.write(b);
			buffer.delete(0, 8);
		}
	}

	public void videEtPadding(ObjectOutputStream out) throws IOException {
		if (buffer!=null && buffer.length()>0) {
			for (int i=buffer.length() ; i<8 ; i++) buffer.append('0');
			int b = Integer.parseInt(buffer.substring(0, 8), 2);
			out.write(b);
			buffer=null;
		}
	}


	private void readObject(java.io.ObjectInputStream in) throws IOException,
                                                        ClassNotFoundException {
		...
	}
```

```
	private final class HuffmanIterator implements Iterator<Boolean> {

		private byte current;
		private byte readed = 8;
		private InputStream is;

		HuffmanIterator(InputStream is) {
			this.is = is;
		}

		public boolean hasNext() {
			if (readed == 8) {
				int lu = -1;
				try {
					if ((lu = is.read()) == -1)
						return false;
				} catch (IOException e) {
					e.printStackTrace();
					return false;
				}
				current = (byte) lu;
				readed = 0;
			}
			return true;
		}

		public Boolean next() {
			hasNext();
			boolean b;
			if ((current & 128) == 128)
				b = true;
			else
				b = false;
			current <<= 1;
			readed++;
			return b;
		}

		public void remove() {
			throw new UnsupportedOperationException();
		}

	}
}
```

5- Question Bonus : Programme de décodage/décodage (4 points)

Ecrire un programme qui permet de coder et/ou décoder un fichier texte mot par mot (utilisez votre classe WordReader réalisée dans un des TP précédents). Le programme sera lancé de la manière suivante :

```
java MonProg [-e ou -d] fichierSource.txt fichierCible.txt
```

-e : encoder
-d : décoder