TP PROG no 5 : Codage de Huffman

1 Introduction

Le codage de Huffman (1952) est une méthode de compression de données qui permet de réduire la taille du codage d'un alphabet. Il substitue à un code de longueur fixe (le code ASCII, par exemple, où chaque caractère est codé sur 8 bits) un code de longueur variable. Ce principe de compression est utilisé dans de nombreuses applications.

Le principe général du codage de Huffman est d'associer à chaque symbole du texte à encoder un code binaire qui est d'autant plus court que le caractère correspondant a un nombre d'occurrences élevé dans le texte à encoder.

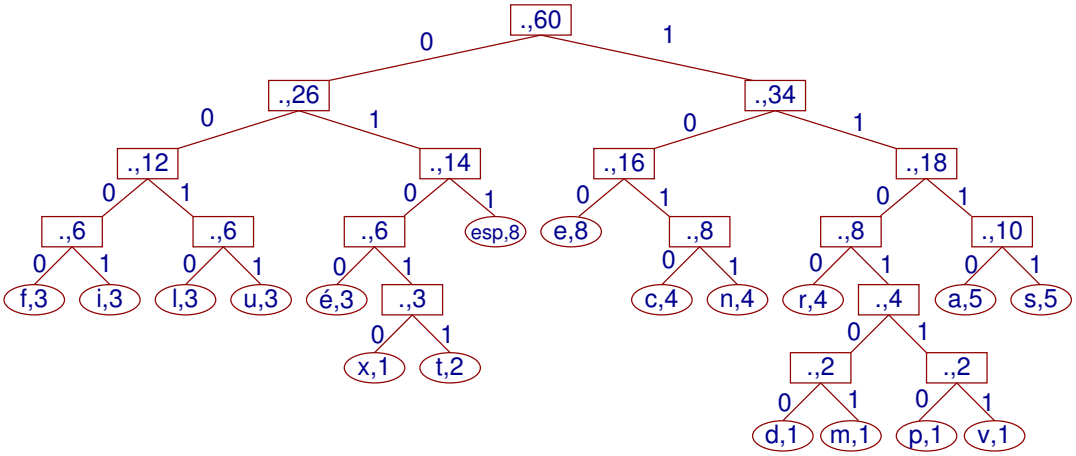
Par exemple, dans le texte "les crépuscules dans cet enfer africain se révélaient fameux" (extrait du « Voyage au bout de la nuit » de Céline), le caractère ' ' (espace) qui apparaît 8 fois pourra être codé avec 3 bits "011", le caractère 'e' qui apparaît 8 fois pourra être codé avec 3 bits "100", le caractère 's' qui apparaît 5 fois pourra être codé avec 4 bits "1111" ; le caractère 'p' qui apparaît 1 fois pourra être codé avec 6 bits "110110", etc. Voici le même texte codé par le codage de Huffman :

```
00101001111011101011000100110110001111111010001100101001111011110100111010111111011101010001011011100101100001001
11000111110000011000001101011100001101101111111000111100010011011101000010111000011001011010110110000111011010110
00001101010
```

La longueur du texte ainsi codé est de 235 bits, qui peuvent se représenter avec 30 octets alors que le texte non codé occupe 60 octets.

Les codes étant de longueur variable, il est nécessaire, pour pouvoir décoder un texte, que le code de chaque caractère ne soit préfixe du code d'aucun autre caractère. Pour obtenir cette propriété, on représente un code de Huffman par un arbre binaire dont les feuilles sont étiquetées par des couples <caractère, fréquence> [1]. Voici l'arbre de Huffman qui permet de coder puis de décoder les caractères du texte ci-dessus :

FIGURE 1 – arbre de (dé)codage de Huffman



Le code d'un caractère est donné par le chemin suivi depuis la racine jusqu'à la feuille où il est situé, en associant un "0" au fils gauche et un "1" au fils droit ; on voit donc que le code du caractère 'f' est "0000", celui du caractère 'x' est "01010" et celui du 'v' est "110111".

1. la valeur des sommets internes n'est pas significative pour le (dé)codage

2 Étapes du codage et du décodage

Cette partie indique les grandes étapes du codage et du décodage ; la réalisation concrète est décrite plus loin (voir § 4 et 5).

2.1 Codage

1. lire le texte contenu dans le fichier à coder ;
2. calculer la fréquence d'apparition de chaque caractère et construire une *table de fréquences* ;
3. enregistrer cette table dans le fichier de sortie ;
4. construire l'*arbre de Huffman* puis l'afficher pour vérification ;
5. construire la *table de codage* associée puis l'afficher pour vérification ;
6. coder le texte avec la table de codage ;
7. enregistrer le texte codé dans le fichier de sortie.

2.2 Décodage

1. lire la table de fréquences dans le fichier à décoder ;
2. construire l'arbre de Huffman puis l'afficher pour vérification ;
3. lire le texte codé ;
4. le décoder avec l'arbre de Huffman ;
5. enregistrer le texte décodé dans le fichier de sortie.

3 Outils fournis

Associez le fichier /share/esir1/prog/05_huffman/outilsHuffman.jar à votre projet ; copiez les autres fichiers du répertoire /share/esir1/prog/05_huffman dans le répertoire de votre projet.

Pour tester vos fonctions *au fur et à mesure de leur achèvement*, vous pouvez utiliser les fonctions fournies dans la classe OutilsHuffman. Pour cela, il suffit de préfixer leur nom par celui de la classe OutilsHuffman. Il vous sera demandé d'écrire vous-même certaines de ces fonctions : pour utiliser vos fonctions à la place de celles fournies, il suffira d'enlever le préfixe OutilsHuffman.

Dans la suite, on appelle *texte codé* une chaîne de caractères composée de '0' et de '1' qui représente un texte dont les caractères sont codés par le codage de Huffman.

fonctions disponibles :

pour le codage :

- static char [ ] lireFichier(String nomFichierACoder) : lit un texte dans un fichier et le renvoie dans un tableau de caractères.
- static void enregistrerTableFrequences(int [ ] tabfreq, String nomFichierCode) : enregistre la table de fréquences dans le fichier codé de sortie.
- static void enregistrerTexteCode(StringBuilder texteCode, String nomFichierCode) : enregistre le texte codé dans le fichier codé de sortie.

pour le décodage :

- static int [ ] lireTableFrequences(String nomFichierCode) : cette fonction lit et renvoie la table de fréquences présente dans le fichier codé de nom donné ; c'est un tableau indicé par les caractères du code ASCII.
- static String lireTexteCode(String nomFichierCode) : lit le texte codé dans le fichier donné et le renvoie dans une chaîne de caractères composée de '0' et de '1'.
- static void enregistrerTexte(StringBuilder texte, String nomFichierDecode) : enregistre le texte décodé dans le fichier indiqué.

pour les deux :

- static ABinHuffman construireArbreHuffman(int [ ] tableFrequences) : construit et renvoie l'arbre de (dé)codage de Huffman qui correspond à la table de fréquence donnée.

divers :

- static ABinHuffman consArbre(Couple<Character, Integer> x, ABinHuffman g, ABinHuffman d) : étant donné un couple <caractère, fréquence> x et deux arbres de Huffman g et d, cette fonction construit et renvoie un nouvel arbre de Huffman de valeur x, dont le fils gauche est g, le fils droit d.
- static long getInstantPresent() : donne un nombre de millisecondes correspondant à l'instant présent ; peut servir pour mesurer le temps que dure chaque traitement.
- static long tailleFichier(String nomFichier) : donne la taille en octets d'un fichier ; sert pour les calculs de taux de compression.

types utilisés dans le TP :

ABinHuffman : arbre binaire de Huffman utilisé pour le (dé)codage ; implémente l'interface ArbreBinaire<T> vue en TD ; l'implémentation est fournie.

Couple<Character, Integer> : type des valeurs d'un arbre binaire de Huffman ; le caractère est celui qu'on veut (dé)coder, l'entier est le nombre d'occurrences du caractère dans le texte à (dé)coder ; l'implémentation est fournie.

ListeABH : liste dont les éléments sont des arbres binaires de Huffman ; utilisée pour construire l'arbre de (dé)codage ; cette liste implémente l'interface List<E> de la bibliothèque java ; *l'implémentation est réalisée par chaînage* (LinkedList) : il n'est donc pas possible d'utiliser les opérations avec indice ; par contre, on dispose de méthodes propres aux listes chaînées comme getFirst, getLast, removeFirst et removeLast par exemple.

String : quelques méthodes ; l'indice d'un caractère dans une chaîne est toujours compris entre 0 et la longueur de la chaîne - 1 :

- char charAt(int idx) : donne le caractère d'indice idx de la chaîne courante.
- int length() : longueur de la chaîne courante.

On rappelle que l'opérateur + permet de créer une nouvelle chaîne par concaténation de deux chaînes ou d'une chaîne et d'un caractère.

StringBuilder : Les chaînes du type String ne sont pas modifiables ; en outre, la concaténation répétée est une opération très lente ; le type StringBuilder est une variante du type String qui permet des opérations de modification très efficaces ; voici quelques méthodes de cette classe :

- StringBuilder() : constructeur ; initialise une chaîne vide ;
- void append(char c) : ajoute le caractère c en fin de chaîne courante.

fichiers de test :

- fichiers de suffixe ".txt" : différents textes à coder ; exemple.txt correspond à l'exemple présenté en introduction.
- fichiers de suffixe ".txt.code" : les mêmes textes codés ; peuvent servir à tester vos fonctions de décodage.

4 Décodage

Compléter la classe DecodageHuffman.

1. Écrire la fonction : static StringBuilder decoderTexte(String texteCode, ABinHuffman arbre) ;
   qui, étant donné un texte codé et l'arbre de Huffman associé renvoie le texte décodé dans une chaîne de caractères de type StringBuilder.
2. Avec les fonctions de la classe OutilsHuffman, *compléter* le programme client (main) pour décoder le contenu du fichier *exemple.txt.code* ; ce programme enchaînera les opérations suivantes (voir aussi § 2.2) :
   - lire la table des fréquences dans le fichier ;
   - construire l'arbre de décodage associé ;
   - lire le texte codé dans le fichier ;
   - décoder le texte et l'afficher au terminal ; vous devriez obtenir la phrase donnée dans l'introduction.
3. Écrire (et tester) dans la classe DecodageHuffman la fonction static void afficherHuffman(ABinHuffman a) ; qui affiche au terminal la valeur de chaque feuille de l'arbre de Huffman (caractère et fréquence), ainsi que le code correspondant.

   Exemple de résultat : Arbre de Huffman associé au texte *exemple.txt.code*

```
<f,3>    : 0000          <c,4>    : 1010
<i,3>    : 0001          <n,4>    : 1011
<l,3>    : 0010          <r,4>    : 1100
<u,3>    : 0011          <d,1>    : 110100
<é,3>    : 0100          <m,1>    : 110101
<x,1>    : 01010         <p,1>    : 110110
<t,2>    : 01011         <v,1>    : 110111
< ,8>    : 011           <a,5>    : 1110
<e,8>    : 100           <s,5>    : 1111
```

Vous pouvez ensuite tester votre décodage sur les différents fichiers fournis.

5 Codage

5.1 Construction de l'arbre de Huffman

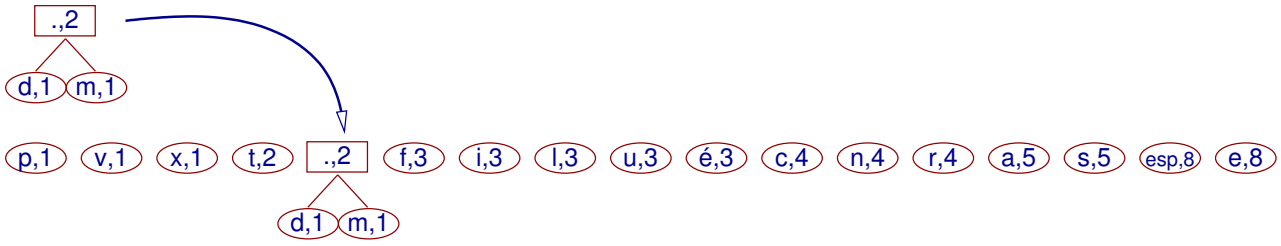
5.1.1 principe

1. À partir du tableau de fréquences et *en ne retenant que les caractères effectivement présents dans le texte*, construire une liste dont les éléments sont des ***arbres binaires de couples <caractère, fréquence>*** (voir figure 2) ; cette liste doit être triée par fréquence croissante.
2. Construire l'arbre de Huffman :
   - prendre les deux éléments (arbres) minimaux de la liste ci-dessus, en faire un *nouvel arbre* dont la racine est un couple dont le caractère est quelconque (sans signification) et la fréquence est la somme des fréquences des deux éléments retirés de la liste.
   - insérer l'arbre obtenu *à sa place* dans la liste triée (voir figure 3) ;
   - répéter jusqu'à ce que la liste ne contienne plus qu'un élément (voir figure 4, page 6) : l'arbre obtenu est l'arbre de (dé)codage associé au texte.

   important : pour que votre arbre final soit compatible avec les fichiers tests, l'insertion dans la liste doit correspondre aux exemples des figures 2, 3 et 4.

FIGURE 2 – liste initiale triée d'***arbres binaires de Huffman***

FIGURE 3 – construction d'un nouvel arbre et insertion dans la liste



5.1.2 réalisation

Compléter la classe CodageHuffman et écrivez les fonctions suivantes ; il n'est pas interdit de réfléchir et d'écrire des fonctions auxiliaires :

1. private static ListeABH faireListeAbinHuffman(int [ ] tableFrequences) : étant donné une table de fréquences, construit et renvoie une liste triée selon le principe décrit au paragraphe 5.1.1.
2. public static ABinHuffman construireArbreHuffman(int [ ] tableFrequences) : construit et renvoie l'arbre binaire de codage de Huffman correspondant à la table de fréquence donnée en paramètre.

Pour tester le bon fonctionnement de la construction de l'arbre de Huffman, il suffit de reprendre votre programme de décodage et d'y remplacer l'appel de la fonction construireArbreHuffman de la classe OutilsHuffman par l'appel de la vôtre.

5.2 Codage du texte

5.2.1 remarque préliminaire

Un caractère est représenté en machine par un entier positif ; dans le cadre de ce TP, les valeurs possibles vont de 0 à 255. La correspondance entre un caractère et sa valeur numérique se fait ainsi :

```
// correspondance caractère → valeur
char c;
int valeur = (int) c;    // 0 ≤ valeur ≤ 255

// correspondance valeur → caractère
int nb;                  // 0 ≤ nb ≤ 255
char c = (char) nb;
```

5.2.2 principe

1. On commence par construire une *table de codage* à l'aide de l'arbre de Huffman : cette table est indicée par les caractères ; pour chaque caractère, la valeur dans la table est :
   - sa chaîne de codage si le caractère est présent dans l'arbre de Huffman,
   - null, si le caractère n'est pas dans l'arbre.

   exemple : voici un extrait de la table de codage associée à l'arbre de Huffman de la figure 1 ; les caractères 'b', 'g' et 'h' qui ne sont pas dans l'arbre n'ont pas de code associé.

indice	...	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	...
chaîne de codage	...	"1110"	null	"1010"	"110100"	"100"	"0000"	null	null	"0001"	...

2. On code ensuite chaque caractère du texte à l'aide de la table de codage.

5.2.3 réalisation

1. Écrire les fonctions suivantes :
   (a) static String [ ] construireTableCodage(ABinHuffman abh) : étant donné un arbre de codage de Huffman, construit et renvoie la table de codage associée.

   Tester cette fonction avec les opérations suivantes :
   - lire le fichier codé
   - calculer la table de fréquences
   - construire l'arbre de Huffman et l'afficher
   - construire et afficher la table de codage

   (b) static StringBuilder coderTexte(char [ ] texte, String [ ] tablecodage) : étant donné un texte à coder disponible dans un tableau de caractères et la table de codage associée, renvoie une chaîne de caractères de type StringBuilder contenant le texte codé.

2. Compléter le programme fourni pour tester l'ensemble du codage (voir § 2.1) :
   - lire le fichier codé
   - calculer la table de fréquences
   - enregistrer la table de fréquences dans le fichier de sortie
   - construire l'arbre de Huffman
   - construire la table de codage
   - lire le texte à coder
   - coder le texte
   - l'enregistrer dans le fichier de sortie

- Tester le codage sur les différents fichiers fournis et vérifier que le décodage produit un fichier identique à l'original... (Sous Linux, on peut comparer deux fichiers texte avec la commande diff et deux fichiers binaires avec la commande cmp).
- Vérifier que votre programme de décodage décode correctement les textes que votre programme de codage a codés.
- Calculer le taux de compression (ratio taille fichier codé / taille fichier non codé) et la durée des différentes opérations.

6 Extensions

Programmer l'opération suivante :

static void enregistrerTexte(StringBuilder texte, String nomFichierDecode) :
enregistre le texte décodé dans le fichier indiqué ; il s'agit d'écrire dans un fichier une chaîne de caractères encodée selon la norme « ISO-8859-1 » [2]

Pour réaliser cette opération, il faudra combiner une instance de chacune des classes ci-dessous (voir l'API java) :
- FileOutputStream is an output stream for writing data to a File or to a FileDescriptor ;
- OutputStreamWriter is meant for writing streams of raw bytes such as image data.
- BufferedWriter writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.

2. voir l'article correspondant sur wikipedia.fr

FIGURE 4 – étapes de la construction de l'arbre de (dé)codage