

GED:

Code Edmodo: 76a89f-

## GED - GLSI - 2016/2017

L'algorithme de Huffman permet une compression de données performante et sans perte. Il est en général utilisé pour compresser du texte par opposition avec les algorithmes avec pertes, comme jpeg, qui sont plutôt employés pour des images. La norme de compression jpeg utilise toutefois le codage de Huffman pour coder certaines informations.

Pour comprimer les données, l'idée est d'utiliser moins de place pour coder les données qui apparaissent souvent que pour celles qui apparaissent plus rarement.

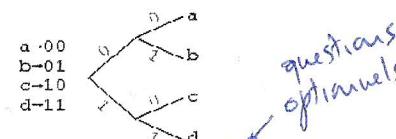
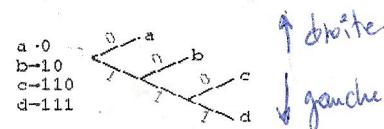
Par exemple pour coder un texte en français, un tel codage utilisera une séquence de bits courte pour coder la lettre 'E' qui apparaît souvent en français et une séquence plus longue pour la lettre 'K' qui apparaît moins souvent (comme dans le code Morse).

Plus précisément, supposons que l'on veuille coder un texte utilisant l'alphabet (a,b,c,d) apparaissant avec les fréquences (8,4,2,2), on utilisera moins de place avec le code binaire (0,10,110,111) qui donnera une longueur de  $8*1+4*2+3*2+3*2=28$  bits qu'avec le code (00,01,10,11) qui donnera une longueur de  $8*2+4*2+2*2+2*2=32$  bits.

### 1 Arbre et code de Huffman

La théorie de l'information fournit une solution optimale pour construire un tel code. Cette solution utilise un arbre binaire complet pour représenter le code. Dans cet arbre, les chemins de la racine vers les feuilles fournissent le codage des lettres de l'alphabet placées dans les feuilles.

Par exemple, la figure suivante donne deux exemples de code et leur représentation sous forme d'arbre.



Question 1, 2, 3 et 4 bis

*Nœud*

✓ Question 1 Pour décrire les arbres binaires, écrire un type Nœud qui a un champ fréquence de type int. Définir dans la classe un constructeur prenant en argument la fréquence d'initialisation.

*Feuille*

✓ Question 2 Pour décrire les nœuds feuilles, définir un type Feuille qui hérite de Nœud et qui y ajoute un champ lettre de type char. Définir dans la classe un constructeur qui prend en argument une

Si vous ne comprenez comment réaliser l'implantation du type des arbres de codes sous forme de types disjonctifs, vous

lettre et sa fréquence.

✓ Question 3 Pour décrire les nœuds internes, définir un type Interne qui hérite lui aussi de Nœud et qui y ajoute deux champs qui contiennent respectivement une référence à un nœud fils gauche (vers le bas sur les figures) et une référence à un nœud fils droit (vers le haut sur les figures). Sachant que la fréquence d'un nœud interne est définie comme la somme des fréquences de ses fils, définir dans la classe un constructeur qui prend en argument un nœud fils gauche et un nœud fils droit.

✓ Question 4 Écrire un test qui construit les deux arbres de la figure précédente avec les fréquences proposées dans l'introduction

✓ remplirTable(...)

✓ Question 5 En vous appuyant sur le mécanisme de redéfinition des méthodes dynamiques, ajouter aux classes précédentes une méthode recursive void remplirTable(String[] codes, String prefix) qui parcourt l'arbre pour construire les codes des lettres sous forme de chaînes de caractères.

✓ remplirTable(Nœud)

Plus précisément, l'appel n.remplirTable(tab, prefix) sur un nœud n de l'arbre des codes complète le tableau de chaînes de caractères tab avec les codes construit à partir du nœud n en les préfixant par prefix. En particulier, sur un objet f de type Feuille contenant par exemple la lettre 'a', l'appel f.remplirTable(tab,"01011") associera le code "01011" à la lettre 'a'

Le code de la lettre i sera placé dans la case i du tableau passé en argument. Par exemple, le code de la lettre char c='a' sera placé dans la case codes[c]. On se limitera à un tableau de taille 256.

Si une lettre n'a pas de code dans l'arbre sa case restera à la valeur d'initialisation du tableau (null).

La méthode remplirTable() de la classe Nœud ne devrait jamais être appelée, elle lèvera donc systématiquement une exception de type Error.

On pourra ajouter à la classe Nœud une méthode de « commodité » static String[] remplirTable(Nœud racine) qui appelle la méthode remplirTable() sur racine en affectant la chaîne vide ("") à prefix.

Compléter le test précédent avec un appel à cette méthode.

(hème + { } de documents  
(objets) qui parlent sur ce hème.  
(dizaine).  
entiers et 10.

pouvez commencer par les planter en utilisant une solution basée sur le champ selecteur.

Écrire une classe Arbre qui contient, en plus du selecteur, les différents champs présents dans les classes Nœud, Feuille et Interne. Ajouter :

- un constructeur pour les feuilles qui prend en argument une lettre et une fréquence ;
- un constructeur pour les nœuds internes qui prend en argument un arbre fils gauche et un arbre fils droit ;
- une méthode dynamique boolean estFeuille() qui permet de déterminer si l'arbre est une feuille.

Si vous adoptez cette solution, il faudra adapter les questions qui suivent à votre implantation.



Pour décoder le code standard des caractères vous pourrez utiliser la méthode suivante que vous aurez préalablement ajoutée à la classe Huffman.

```
static char decodeASCII(Etat e) {
    if (!e.valide()) {
        throw new Error("Fin de code atteinte");
    }
    char c = 0;
    for (int j=128;j>0;j=j>>1) {
        if (e.nextBit()) {
            c|=j;
        }
    }
    return c;
}
```

*Arrng (Binar) → char.*

*Exercice ?*  
**Question 12 (optionnelle)** Dans la pratique le texte est directement codé en binaire octet par octet ; pas dans une chaîne de caractères. La taille du codage produit doit donc être un multiple de 8. Pour assurer cela, l'idée consiste à ajouter en début de code des '1' qu'il est facile de différencier du début du codage de l'arbre (il commence toujours par '0').

En vous appuyant sur le mécanisme de redéfinition des méthodes dynamiques et en utilisant les fréquences stockées dans l'arbre, ajouter aux classes utilisées pour représenter l'arbre des codes une méthode int tailleCodage(int profondeur) qui retourne la taille du codage de l'arbre suivi de celui du texte.

En utilisant ces méthodes, écrire une méthode complète d'encodage et de décodage.

### 3 Construction du code

Maintenant que nous avons écrit les fonctions de codage et de décodage à partir d'un arbre de codes, il nous reste à construire ce dernier en fonction de la fréquence des lettres dans le texte initial.

*b1+b2*  
*b1* ↗  
*b2* ↘  
**Question 13** L'algorithme de construction de l'arbre des codes, requiert de pouvoir sélectionner un arbre (de type Noeud) de plus petite fréquence parmi un ensemble d'arbres.

Ajouter dans la classe Noeud des méthodes boolean moinsPrioritaireQue(Noeud n) et boolean plusPrioritaireQue(Noeud n) qui font des comparaisons strictes sur les fréquences des arbres. Le noeud le moins fréquent est le plus prioritaire.

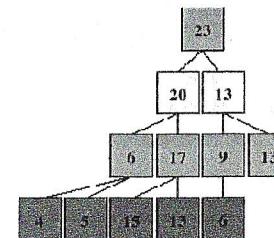
*✓ Question 14* Une structure de données bien adaptée pour le problème d'obtention du plus petit (ou de plus grand) élément d'un ensemble est le tas aussi appelé file de priorité.

Un tas est un arbre binaire qui respecte les propriétés suivantes :

- tous les niveaux sont remplis sauf éventuellement le dernier qui est rempli à gauche.  
On dit que l'arbre est tassé;

- tout noeud du tas a une « priorité » plus forte ou égale à celle de ses fils. Ainsi, la racine contient le noeud de plus forte priorité.

L'arbre suivant respecte la propriété d'un tas.



Dans notre cas, chaque noeud du tas fera référence à un objet de type Noeud dont la fréquence devra être plus faible que celles des objets Noeud référencés par ses fils. La priorité sera donc l'inverse de la fréquence.

Une manière compacte de représenter un arbre binaire tassé consiste à utiliser un tableau dont chaque case correspond à un noeud de l'arbre et telle que pour toute case d'indice i :

- la case d'indice  $(i-1)/2$ , si elle est valide, contient le noeud père du noeud d'indice i ;
- la case d'indice  $2*i+1$ , si elle est valide, contient le fils gauche du noeud d'indice i ;
- la case d'indice  $2*(i+1)$ , si elle est valide, contient le fils droit du noeud d'indice i.

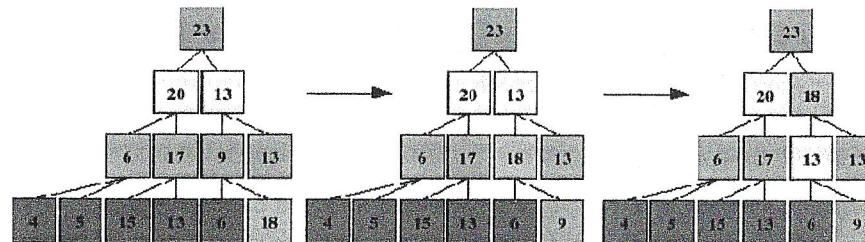
En utilisant une telle implantation, l'arbre de l'exemple précédent est représenté de la façon suivante :

0	1	2	3	4	5	6	7	8	9	10	11
23	20	13	6	17	9	13	4	5	15	13	6

Définir une classe Tas permettant de représenter un tas. Cette classe contient :

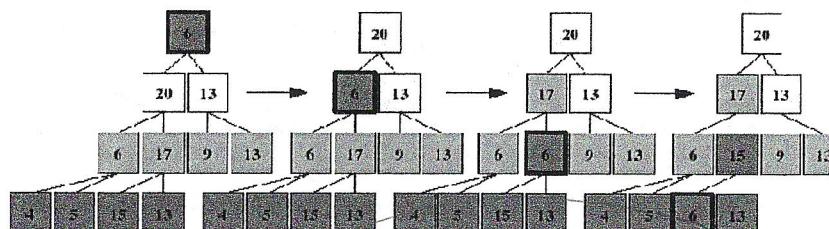
- un champ noeuds qui référence un tableau de Noeud ;
- un champ nb qui correspond au nombre d'éléments dans le tas (toujours inférieur ou égal à la taille du tableau) ;
- un constructeur sans argument (Tas()) qui initialise nb à 0 et le tableau référencé par noeuds à une taille par défaut (par exemple 1 pour les tests) ;
- une méthode boolean singleton() qui retourne true si le tas contient un seul élément ;
- éventuellement des méthodes de commodité static int filsGauche(int i), static int frereDroit(int i) et static int pere(int i).

*✓ Question 15* On souhaite maintenant pouvoir ajouter un élément dans le tas. Pour cela, on ajoute le nouvel élément en fin de tableau. S'il n'y a plus de place dans le tableau, on double sa taille en créant un nouveau tableau et en y recopiant l'ancien. Ensuite, si cela est nécessaire pour respecter la propriété du tas, on fait remonter l'élément inséré dans l'arbre, comme illustré par la figure suivante où un élément de priorité 18 a été inséré dans l'arbre de notre premier exemple :



Ajouter à la classe Tas une méthode void ajouter(Noeud nouveauNoeud) qui ajoute un élément dans le tas.

- ✓ **Question 16** On souhaite maintenant pouvoir retirer un élément du tas. Pour cela, on retire l'élément de tête que l'on remplace par l'élément en fin de tableau. Dans notre exemple, qui repart de l'arbre de notre premier exemple, l'élément racine 23 est remplacé par la feuille 6. Ensuite, si cela est nécessaire pour respecter la propriété du tas, on fait descendre l'élément inséré dans l'arbre, comme illustré par la figure suivante :



Ajouter à la classe Tas une méthode Noeud retirer() throws TasVideException qui retire un élément du tas et le retourne. Si le tas est vide, la méthode lèvera une exception de la classe TasVideException que vous aurez préalablement définie comme héritant de la classe Exception.

- ✓ **Question 17** Dans la classe Huffman, écrire une méthode static Feuille[] initialiseFrequence(String texte) qui retourne un tableau tel que la case d'indice *i* contient null si le caractère de code ASCII *i* n'apparaît pas dans texte et qui contient un objet de type Feuille dont le champ lettre est initialisé avec *i* et dont le champs fréquence est égal au nombre d'apparitions de la lettre de code ASCII *i* dans texte.

- ✓ **Question 18** Maintenant, il reste à planter la construction de l'arbre des codes. L'algorithme est le suivant : tant qu'il y a plus d'un élément dans le tas retirer les deux objets de type Noeud de plus petite fréquence du tas ; construire un nouvel objet de type Interne dont les fils sont les deux Noeud qui viennent d'être retirés et l'ajouter au tas. Le dernier noeud qui reste dans le tas est l'arbre des codes.

Dans la classe Huffman, ajouter une méthode static Noeud construitCode(Feuille[] lettres) throws TasVideException qui construit un arbre de code à partir d'un tableau de feuilles construit avec la méthode initialiseFrequence().

Faire en sorte que vous puissiez coder et décoder un texte ne contenant qu'une seule lettre comme par exemple "aaaaaaaaaaaaaaaa".

Dans la classe Huffman, ajouter une méthode de codage complète.

- ✓ **Question 19 (optionnelle)** L'algorithme de compression travaille dans la réalité sur des fichiers plutôt que sur des chaînes de caractères. La classe suivante présente un exemple qui recopie le contenu d'un fichier dont le nom est passé en premier argument dans un autre dont le nom est passé en deuxième argument. La fonction de lecture manipule des caractères qui sont stockés dans des int pour pouvoir coder, en plus des caractères, l'indication de la fin du fichier (-1).

```
import java.io.*;
class Copy {
    public static void main(String[] args) throws IOException {
        if(args.length != 2) {
            System.err.println("Usage: java Copy from to");
            System.exit(1);
        }
        FileReader reader = new FileReader(args[0]);
        FileWriter writer = new FileWriter(args[1]);
        while(true) {
            int c = reader.read();
            if(c == -1) {
                // Si la fin de fichier est atteinte
                break;
            }
            writer.write(c);
        }
        reader.close();
        writer.close();
    }
}
```

Vous aurez également besoin des opérateurs de manipulation binaire *et* (&), *ou* () et *décalage à droite des bits* (>>) utilisées dans la fonction encodeASCII() et decodeASCII() précédentes. Écrire une commande de compression et de décompression de fichiers basée sur le codage de Huffman.

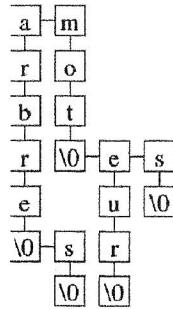
- ✓ **Question 20 (optionnelle)** Le tas est une structure de données qui permet d'implanter un algorithme de tri efficace.

Dans la classe Tas, écrire une méthode de tri d'un tableau de feuilles.

#### 4. Compression de dictionnaire

On souhaite maintenant compresser un dictionnaire. Pour cela, plutôt que de compresser les mots séparés par des espaces avec l'algorithme de Huffman, nous vous proposons d'abord de compresser le dictionnaire en le représentant sous la forme d'un arbre qui permet de partager les préfixes (début) des mots, puis de compresser les lettres apparaissant dans les nœuds de l'arbre du dictionnaire avec l'algorithme de Huffman.

Supposons que le dictionnaire soit composé des mots arbre, arbres, mot, moteurs et mots, l'arbre représentant le dictionnaire est alors décrit par la figure suivante :



Pour marquer la fin des mots, on utilise le caractère spécial '\0' qui normalement n'est pas utilisé pour coder les mots.

✓ **Question 21** Écrire une classe `NoeudDico` qui permet de représenter un nœud de l'arbre du dictionnaire.

✓ **Question 22** Ajouter à la classe `NoeudDico` un constructeur récursif `NoeudDico(String mot, int i)` qui permet de construire un arbre représentant le dictionnaire contenant un seul mot : le suffixe du mot mot à partir de la lettre d'indice i (inclus). On supposera que la chaîne de caractères mot se termine par le caractère '\0' qui aura préalablement ajouté à la fin du mot.

Par exemple, `new NoeudDico("arbre\0", 2)` construit le dictionnaire contenant le suffixe "bre\0".

✓ **Question 23** Ajouter à la classe `NoeudDico` un constructeur simple qui prend en argument une lettre, un nœud fils et un nœud frère.

✓ **Question 24** Ajouter à la classe `NoeudDico` une méthode dynamique récursive `NoeudDico ajouter(String mot, int i)` qui ajoute le suffixe du mot mot à partir de la lettre d'indice i dans le dictionnaire courant. Le dictionnaire courant est éventuellement modifié et le nouveau dictionnaire est retourné par la méthode.

Testez votre fonction en créant un dictionnaire à partir, par exemple, du dictionnaire de l'anglais : `/usr/dict/words`. Pour cela, vous pourrez utiliser la méthode `TC.motsDeFichier()`.

✓ **Question 25** Ajouter à la classe `NoeudDico` une méthode d'affichage du dictionnaire.

✓ **Question 26** En vous inspirant du codage binaire de l'arbre des codes et en utilisant un codage de Huffman pour coder les lettres présentent dans l'arbre du dictionnaire, écrire une commande d'écriture et de lecture de dictionnaire compressé.

## 5 Huffman sur les mots

Maintenant que nous disposons d'un moyen de compresser un dictionnaire, l'idée de l'algorithme de compression de Huffman sur les mots est de construire le dictionnaire des mots d'un fichier, puis de coder chaque mot en fonction de sa fréquence en utilisant l'algorithme de Huffman.

✓ **Question 27** Pour pouvoir simplement appliquer l'algorithme de Huffman, il faut associer un entier à chaque mot du dictionnaire, et inversement.

Proposer une implantation de telles fonctions, puis modifier l'algorithme de Huffman sur les lettres pour qu'il fonctionne sur les mots.

✓ **Question 28** Réaliser un compresseur/décompresseur de fichiers texte utilisant l'algorithme de Huffman sur les mots.

## 6 Huffman adaptatif

La méthode de compression de Huffman a deux inconvénients :

- il faut lire le texte entièrement avant de lancer la compression ;
- il faut aussi transmettre le code.

Une version adaptative de l'algorithme de Huffman corrige ces défauts, son principe est le suivant :

L'arbre initial est constitué d'une unique feuille, celle de la lettre vide. À chaque fois qu'un caractère est lu dans le texte source :

- s'il est déjà apparu :
  - on imprime son code,
  - on met à jour l'arbre,
- sinon
  - on imprime un code spécial suivi du code ASCII de la nouvelle lettre ;
  - on ajoute une feuille dans l'arbre ;
  - on met à jour l'arbre.

Évidemment, le code d'une lettre change en cours de transmission : quand la fréquence (relative) d'une lettre augmente, la longueur de son code diminue. Le code d'une lettre s'adapte à sa fréquence. Le destinataire du message décode le codage de l'expéditeur : il maintient l'arbre de Huffman qu'il met à jour comme l'expéditeur, et il sait donc tout moment quel est le code d'une lettre.

✓ **Question 29** Proposer une implantation de l'algorithme de Huffman adaptatif.