ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  
UNIVERSITÉ DU QUÉBEC

LOG320 – LABORATOIRE #1  
ALGORITHME DE HUFFMAN

PAR  
MARC-ANDRÉ DESTREMPES – AJ86290  
SAMUEL BEAUCHEMIN – AK06890

MONTRÉAL, 23 mai 2013

Table des matières

[Description de votre programme 3](#_Toc356590736)

[Analyse de complexité des algorithmes 4](#_Toc356590737)

[Description des problèmes rencontrés 5](#_Toc356590738)

[Description des améliorations que vous avez implémentées 6](#_Toc356590739)

# Description de votre programme

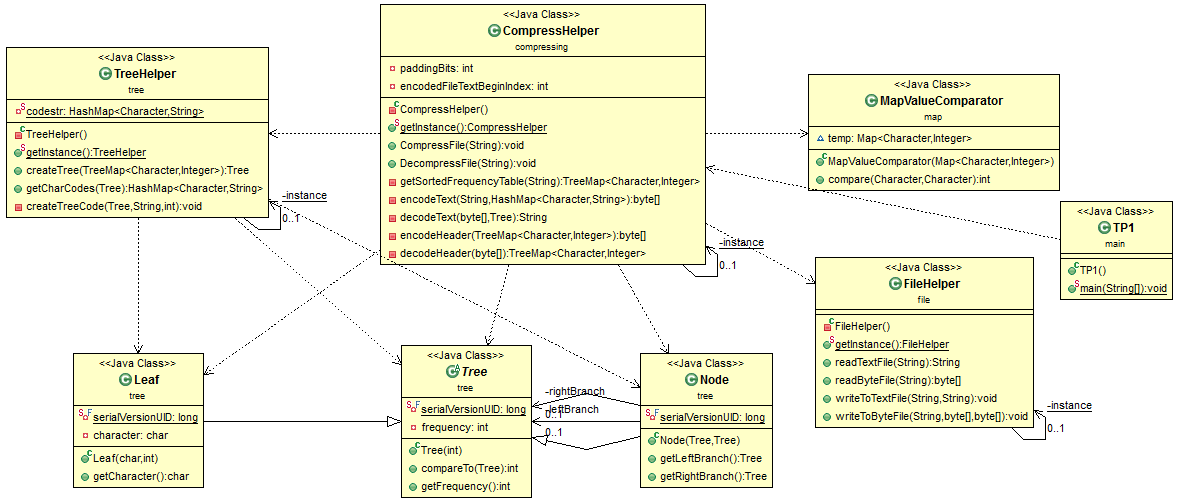


Figure - Diagramme de classe

Le programme se divise en 2 sections, l’encodage et le décodage. L’encodage est ce qui permet de compression un fichier en utilisant l’algorithme de Huffman tandis que le décodage permet de faire l’opération inverse. L’arbre de fréquence est stocké dans un objet *Tree* qui contient des *Node* ou des *Leaf*. Les *Leaf* représentent chacune des lettres avec leur fréquence, tandis que les *Node* sont l’intersection de soit deux *Node* ou deux *Leaf* ou un mixte des deux et contient l’addition de la fréquence des deux classes. La classe *MapValueComparator* permet de trier une map par valeur au lieu de par les clés. La classe *FileHelper* permet de gérer toutes les opérations qui ont rapport avec les fichiers textes, soit lire et écrire, la classe *TreeHelper* permet de créer un arbre, le code ou bien d’avoir le caractère d’un code et la classe *CompressHelper* permet d’effectuer les opérations pour compresser ou décompresser un fichier.

L’encodage commence par lire le fichier voulu. Ensuite, il construit l’arbre et il construit le code représentant chacune des lettres. Par la suite, il remplace chacun des caractères ASCII par le code correspondant construit précédemment. Pour terminer, il écrit le fichier en stockant la fréquence de chacune des lettres dans le header et le nouveau contenu du fichier dans le payload.

Le décodage comme par lire le fichier compressé et sépare le header ainsi que le payload. À partir du header, on reconstruit l’arbre de fréquence qui nous permettra de reconstruire le contenu du payload. Après, on parcourt l’arbre pour décoder le contenu du payload et le retransformer en texte. Pour finir, lorsque toutes les opérations de décodage ont été faites, on réécrit le fichier original.

Le programme a comme limitation les 256 premiers caractères ASCII[[1]](#footnote-1).

# Analyse de complexité des algorithmes

|  |  |
| --- | --- |
| Module.Méthode | Complexité |
| TreeHelper.getInstance() |  |
| TreeHelper.createTree() |  |
| TreeHelper.getCharCode() |  |
| TreeHelper.createTreeCode() |  |
| Leaf.getCharacter() |  |
| Tree.getFrequency() |  |
| Tree.compareTo() |  |
| Node.getLeftBranch() |  |
| Node.getRightBranch() |  |
| MapValueComparator.compare() |  |
| FileHelper.getInstance() |  |
| FileHelper.readTextFile() |  |
| FileHelper.readByteFile() |  |
| FileHelper.writeToTextFile() |  |
| FileHelper.writeToByteFile() |  |
| CompressHelper.getInstance() |  |
| CompressHelper.CompressFile() |  |
| CompressHelper.DecompressFile() |  |
| CompressHelper.getSortedFrequencyTable() |  |
| CompressHelper.encodeText() |  |
| CompressHelper.decodeText() |  |
| CompressHelper.encodeHeader() |  |
| CompressHelper.decodeHeader() |  |

Tableau 1 - Complexité des méthodes

# Description des problèmes rencontrés

Un premier problème rencontré a été l’écriture du fichier lors de l’encodage. Le header du fichier s’écrit correctement, mais le payload ne s’écrit pas en binaire, il écrit le binaire comme un string. Alors, au lieu de compresser le fichier, le fichier augmente de taille car tout est écrit comme un string. Le problème a été réglé lors de la réécriture complète de notre algorithme de compression.

Un deuxième problème rencontré a été la limitation du nombre de fréquence d’un caractère. Le nombre maximal initial était de 256 pour un même caractère. Le problème a été réglé en changeant la façon dont l’information était stockée dans le header.

Un troisième problème rencontré a été l’information à stocker dans le header du fichier pour minimiser l’espace que celui-ci occupe tout en maximisant l’information. Nous avons dont opter pour les fréquences de chacun des caractères.

Un quatrième problème rencontré a été quelle structure de données utilisé pour construire l’arbre. Une première version a été faite avec seulement des *Node* qui contenaient toute l’information. Cette version avait des problèmes lorsqu’il venait le temps de parcourir l’arbre en entier. Nous avons donc décidé d’en venir à une deuxième version, celle qui est contenu dans la Figure 1 - Diagramme de classe, qui contient une superclasse *Tree* et des enfants *Node* et *Leaf*.

# Description des améliorations que vous avez implémentées

Des améliorations… lesquels LOL

1. <http://www.asciitable.com/> [↑](#footnote-ref-1)