Projet SDD

Algorithme de Huffman pour la compression et la décompression de données

Wissam AIT KHEDDACHE
Louka DOZ

Le 15 mai 2021

Introduction

Le présent rapport représente le travail que nous avons fait pour réaliser le projet qui a pour objectif le codage de l'algorithme de Huffman ainsi que les méthodes de compression et de décompression, en utilisant le langage de programmation Java.

Dans ce qui suit, nous allons présenter les fonctionnalités que nous avons mises en œuvre, décrire les structures de données utilisées et les résultats obtenus pour les trois fichiers qui ont été donnés.

manuel d'utilisation

Le main du projet se trouve dans *src/FileCompressor.java*. Lancez-le sans argument pour lancer le programme.

Le programme va d'abord demander ce que vous voulez faire. Entrez "compress", pour compresser un fichier, ou "decompress", pour décompresser un fichier.

L'étape suivante consiste à renseigner le fichier à compresser/décompresser. Un chemin absolu ou relatif peut être renseigné. Par exemple, pour compresser le fichier1.txt, écrivez original/fichier1.txt.

Ensuite, il faut renseigner le dossier qui va recevoir le fichier compressé/décompressé. Un chemin absolu ou relatif peut être renseigné.

Entrez "//quit" pour quitter à tout moment.

Implémentation du HeapSort

Afin de pouvoir construire un arbre de Huffman à partir d'un fichier donné, nous devons avoir au début, une liste de fréquences (nombre d'occurrences) de tous les caractères du fichier triée par ordre croissant.

Il existe plusieurs manières de tri d'un tableau, à savoir le BubbleSort, le QuickSort et puisque on nous a demandé de le trier nécessairement avec le HeapSort, nous avons procédé à son implémentation en créant la classe $FrequencyReader$, qui se charge de :

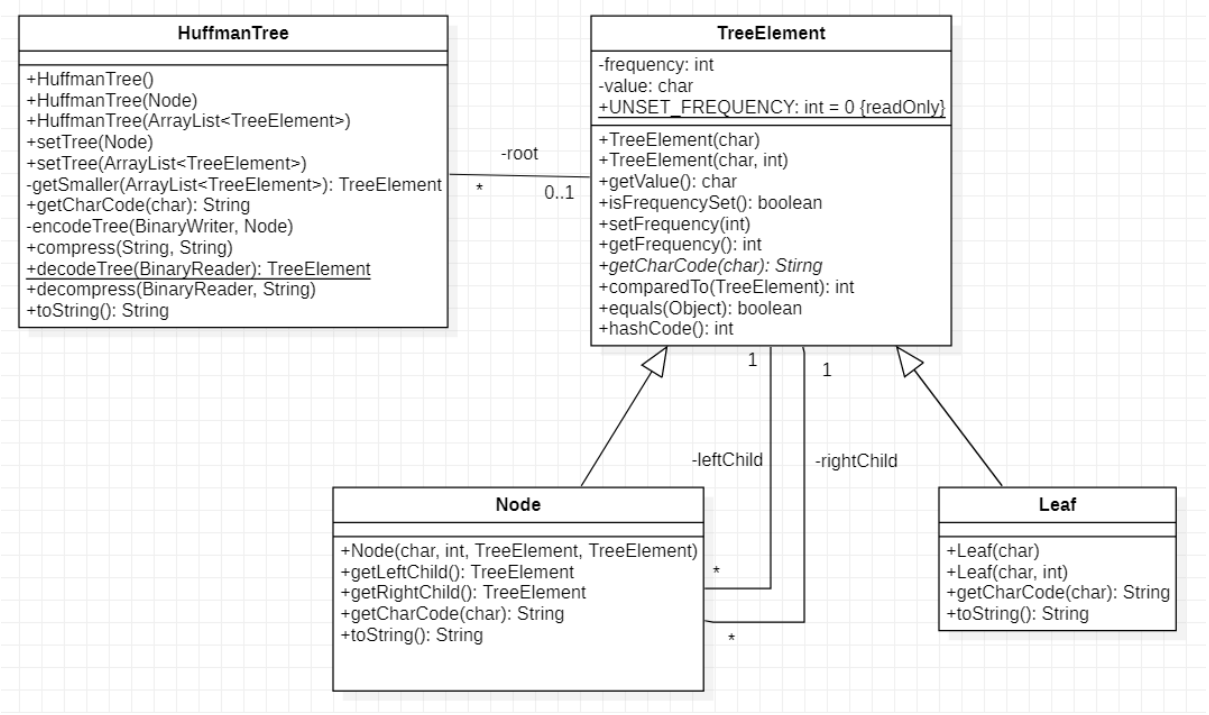
- La lecture du fichier en entrée et remplir le tableau de fréquences de ses caractères et ce, avec la méthode $ArrayList < TreeElement > readFile(String\ file)$ qui prend le chemins vers le fichier en paramètres et retourne un $ArrayList < TreeElement >$ contenant des objets de type $TreeElement$ qui représente les éléments de l'arbre de Huffman par la suite et qui a comme attributs les caractères et leurs fréquences.

- Construire le tas avec la méthode $ArrayList < TreeElement > buildHeap(ArrayList < TreeElement >)$ qui fait appel à une méthode auxiliaire $void\ heapifyTree(ArrayList < TreeElement > frequencyArray, int\ size, int\ index)$ qui a pour but de construire des sous tas récursivement.
  Le tas sera retourné sous forme d'un $ArrayList < TreeElement >$

- Enfin, envoyer le tas construit en paramètres de la méthode $ArrayList < TreeElement > heapSort(ArrayList < TreeElement >)$ afin de le trier par ordre croissant et retourner une copie de l'$ArrayList$ triée.

L’arbre de Huffman

Dans notre projet, nous avons mis au point la structure suivante, pour représenter l’arbre de Huffman :



Structure de l’arbre de Huffman

Les feuilles sont les extrémités de l’arbre. Elles contiennent forcément une valeur et une fréquence qui peut ne pas être définie, elle sera alors automatiquement mise à 0.
Les nœuds (*Node*) ont forcément deux fils qui peuvent être des feuilles (*Leaf*) ou d’autres nœuds, une valeur (celle du fils de droite), et une fréquence (la somme de celle des fils).

La classe *HuffmanTree* désigne un arbre, dont la racine est l’attribut “root” de type *Node*, et toutes les méthodes que l’on peut utiliser avec cet arbre (seule *decodeTree(BinaryReader)* fait exception car elle ne dépend pas de l’arbre, d’où le fait qu’elle soit statique).

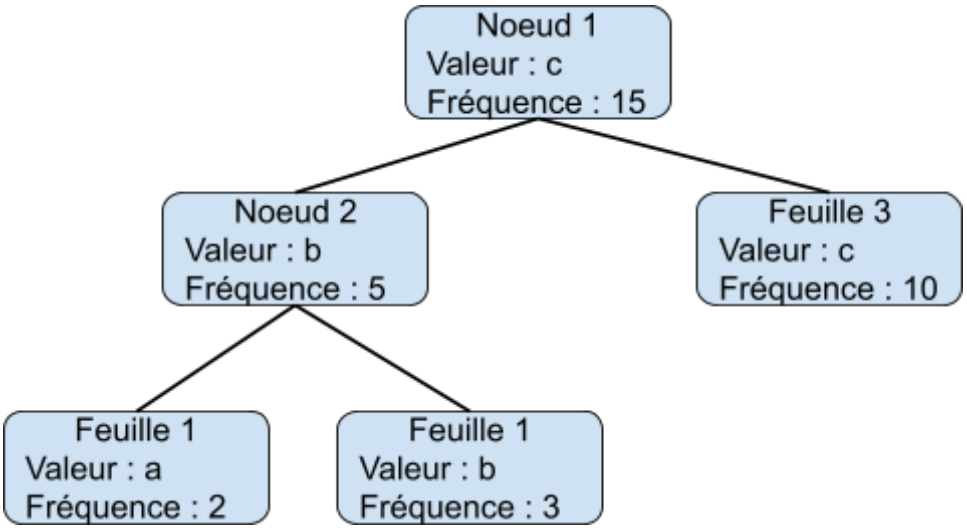
Stockage de l'arbre en début de fichier

Pour stocker l'arbre au début du fichier compressé, nous avons fait le choix de parcourir l'arbre en profondeur et d'ajouter :

- 0 lorsque l'on croise un nœud. Les enfants des noeuds sont parcourus en privilégiant celui de gauche;
- 1 lorsque l'on croise une feuille, suivi de la valeur en binaire sur 16 bits.

De cette manière là, on obtient une suite de 0 et de 1, comme le contenu du fichier après l'avoir encodé avec l'arbre de Huffman, ce qui permet d'appliquer la même compression : découper la suite du bits en octets et écrire les octets un à un.

Prenons l'arbre suivant en exemple :



Pour encoder l'arbre, on part de la racine (Noeud 1), qui est un nœud donc on ajoute 0. Puis on continue sur l'enfant de gauche (Noeud 2), aussi un nœud donc on ajoute 0. Encore une fois on choisit l'enfant de gauche (Feuille 1), qui est une feuille et donc on ajoute 1, mais on ajoute aussi 0000000000110000, qui correspond au caractère 'a' en binaire. Puisque l'on est sur une feuille, il n'y a plus moyen de continuer donc on retourne sur le Noeud 2 et on part à droite et ainsi de suite jusqu'à ce que tout l'arbre soit totalement parcouru.

Finalement, on obtient : 0010000000001100001100000000011000101000000000110001 1.
Noeud Feuille _____ a _____/ _____ b _____/ _____ c _____/

Avec cette méthode de stockage, il n'y a pas besoin de marquer une délimitation entre l'arbre et le contenu. En effet, à la décompression, l'arbre reconstruit est complet lorsque tous les nœuds ont leurs deux enfants définis.

Bilan

Pour conclure, l'ensemble des fonctionnalités des questions 2 à 5 ont été implémentées. Les résultats obtenus sont :

fichier1.txt :

- taille d'origine : 1166 octets
- taille après compression : 894 octets
- taux de compression : 0.7667239 (76.672386 %)
- code de 'A' = 00001
- code de 'B' = 11010010
- code de 'C' = 1110000

fichier2.txt :

- taille d'origine : 17170134 octets
- taille après compression : 7596228 octets
- taux de compression : 0.44240937 (44.240936 %)
- code de 'A' = 000001
- code de 'B' = 1001111101
- code de 'C' = 100110101

fichier3.txt :

- taille d'origine : 12298223 octets
- taille après compression : 8373478 octets
- taux de compression : 0.6808689 (68.08689 %)
- code de 'A' = 1001101
- code de 'B' = 1101001
- code de 'C' = 1000100

Cependant, le programme possède un défaut de performances. Nous voulions utiliser *BuffuredWriter.write(char)* ou *PrintWriter.print(char)* pour écrire les octets, mais les fichiers compressés étaient extrêmement lourds, voire plus lourds que les fichiers d'origine. La solution trouvée a été d'utiliser *FileOutputStream.write(byte)* mais en contrepartie, le vitesse d'exécution est ralentie.