Rapport de Projet de Programmation Impérative

Compression et Décompression de fichiers utilisant l'algorithme de Huffman

Clément Delmaire-sizes et Priscilia Gonthier

Groupe MN02

15 Janvier 2022

Département de Sciences du numérique - Première Année

2021-2022

Résumé

Ce rapport a pour objectif d'informer sur l'organisation et les méthodes adoptées afin de concevoir des algorithmes capables de compresser et décompresser des fichiers à l'aide du codage de Huffman.

Ce document comprend alors une introduction présentant le plan de ce dernier et le problème traité dans ce projet ainsi que l'architecture en module des programmes principaux. Il comprendra aussi une présentation des choix réalisés, des algorithmes et types de données importants et de la démarche adoptée pour tester ces algorithmes. Le rapport détaillera de plus les problèmes rencontrés lors de la conception des algorithmes et les solutions trouvées pour les contourner. Enfin, il expliquera l'organisation de notre équipe, un bilan technique du projet ainsi que le bilan global du projet.

Introduction

Le problème principal de notre projet est de créer un programme de compression et un programme de décompression utilisant l'algorithme de Huffman. Le codage consiste à associer aux caractères du fichier des suites de bits d'autant plus courtes que le caractère est fréquemment présent dans le texte, permettant alors un gain d'espace significatif. Nos programmes devaient pouvoir compresser et décompresser plusieurs fichiers sur une même ligne de commande. Il devaient aussi, si l'utilisateur utilise l'option bavard, afficher certaines des étapes de compression et décompression. Nos programmes devaient aussi être robuste et traiter les différents cas de mauvaise utilisation de nos programmes.

Pour commencer notre rapport détaille ensuite les principaux algorithmes ainsi que les types de données utilisés, puis dans un second temps il explique certains choix que nous avons dû faire. Il présente l'architecture de nos programmes, puis détaille la démarche de test de nos programmes. Vous trouverez ensuite dans ce document les difficultés que nous avons rencontrées et les solutions que nous avons adoptées ainsi que l'organisation de notre équipe tout au long du projet. Il se termine par un bilan technique du projet et un bilan personnel et individuel de celui-ci.

Table des matières:

I. Algorithmes et types

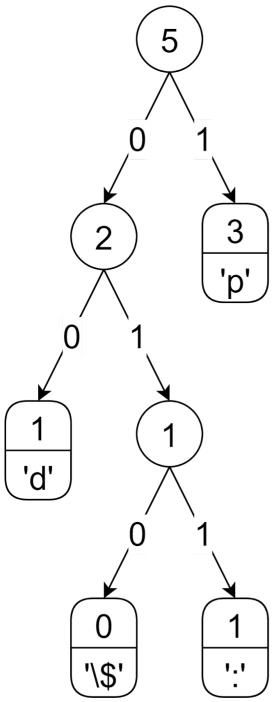
a. Présentation des types

La conception de nos programmes nous a amené à créer 3 types principaux.

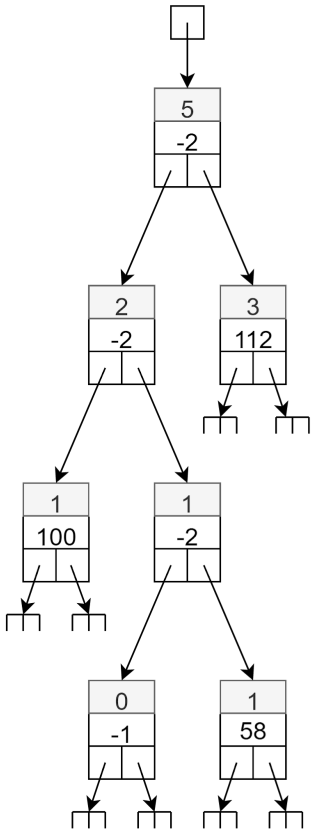
Le premier est le type T_AB, c'est un arbre de Huffman qui associe à chaque caractère sa fréquence d'apparition dans le fichier à compresser. On a décidé pour ce faire d'utiliser un chaînage hiérarchique. Il a été réalisé grâce à un pointeur qui pointe vers une cellule. Cette cellule est un enregistrement de la fréquence du caractère, du caractère en lui-même (codé en Latin1 ou -1 pour /$), puis de 2 sous-arbres qui sont des pointeurs vers d'autres cellules. Les feuilles de l'arbre contiennent les caractères du fichier et les branches ont un caractère qui ne code rien (nous avons choisi arbitrairement -2) ainsi que la somme des fréquences des sous-arbres.

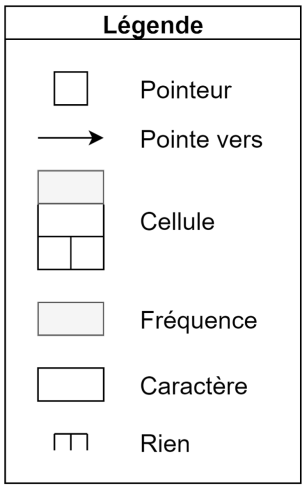
<u>Schéma d'un arbre ainsi que la représentation de notre type</u>

Arbre de Huffman modélisé :



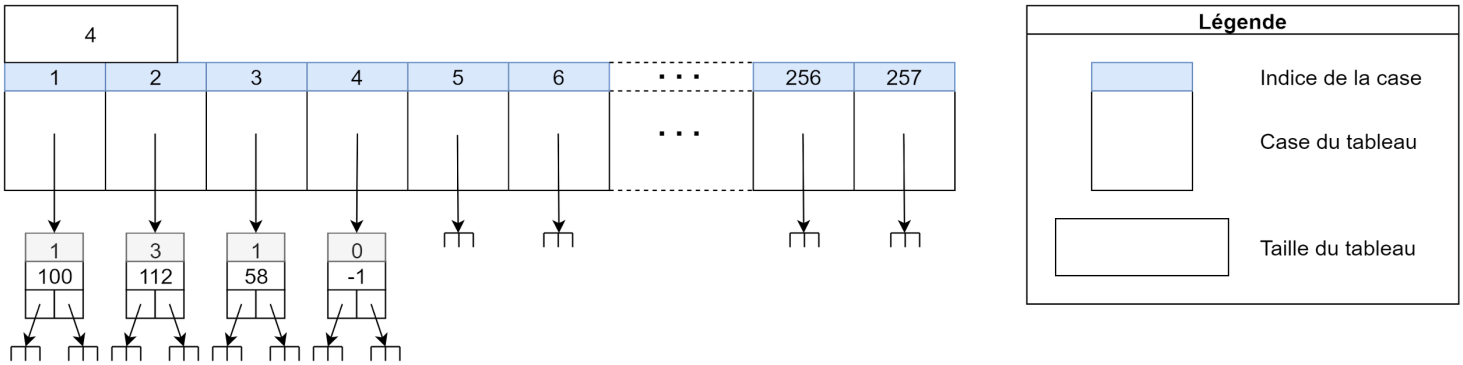
Représentation de l'Arbre de Huffman via notre type T_AB :





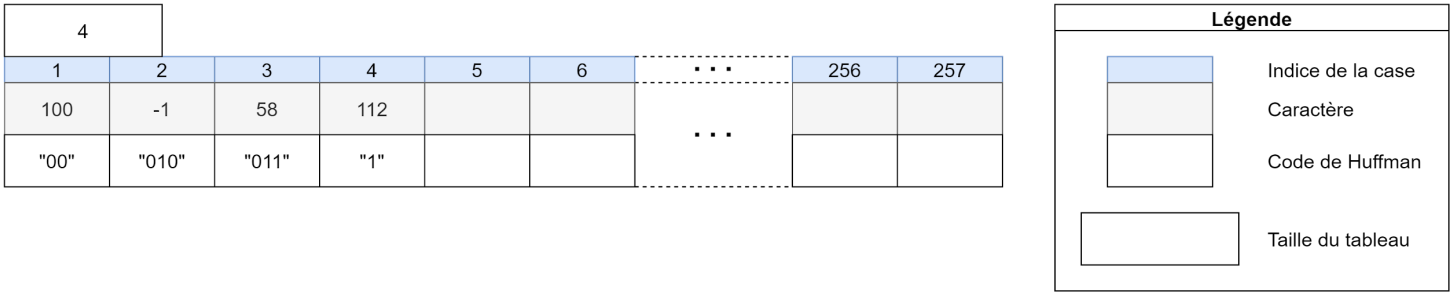
Le second type est le type T_TAB_AB, c'est un tableau d'arbre de taille variable mais possédant une capacité maximale de 257 (car il y a 256 caractères possibles plus le caractère /$). Il est composé d'un enregistrement d'un tableau de 1 à 257 de T_AB et de la taille effective du tableau.

Exemple d'un tableau d'arbre de taille 4 contenant des arbres unitaires (avec 1 seul caractère)



Le dernier type est T_TAB_HUFF, c'est la table de Huffman qui contient les caractères avec leur code de huffman associé. C'est un enregistrement d'un tableau de 1 à 257 de cellule_code et de la taille effective du tableau. Cellule_code est quant à lui un enregistrement du caractère en lui-même (codé en Latin1 ou -1 pour /$) et du code de Huffman associé à ce caractère de type Unbounded_String.

Exemple de Table de Huffman



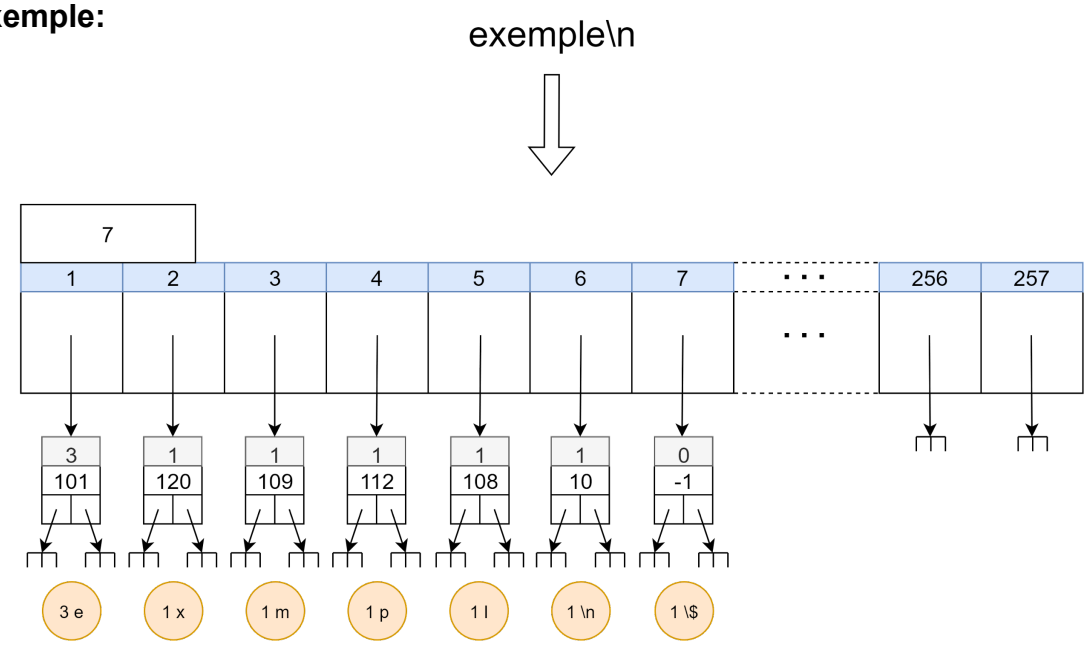
b. Présentation de l'algorithme de compression

Pour présenter l'algorithme nous suivrons un exemple fil-rouge qui nous permet d'imager nos propos.

Le fil-rouge est un fichier contenant le mot exemple (il contient aussi un saut de ligne final car lors de l'écriture d'un fichier il y a toujours un saut de ligne inséré avant la fin du fichier).

En premier lieu le programme parcourt le fichier et stocke les caractères associés à leur fréquence dans un T_TAB_AB. Pour cela, tant que la fin du fichier n'est pas atteinte, il insère le caractère dans ce T_TAB_AB et incrémente la fréquence d'apparition de 1. puis il insère le caractère \$ de fréquence 0 qui va servir à coder la fin du fichier compresser.

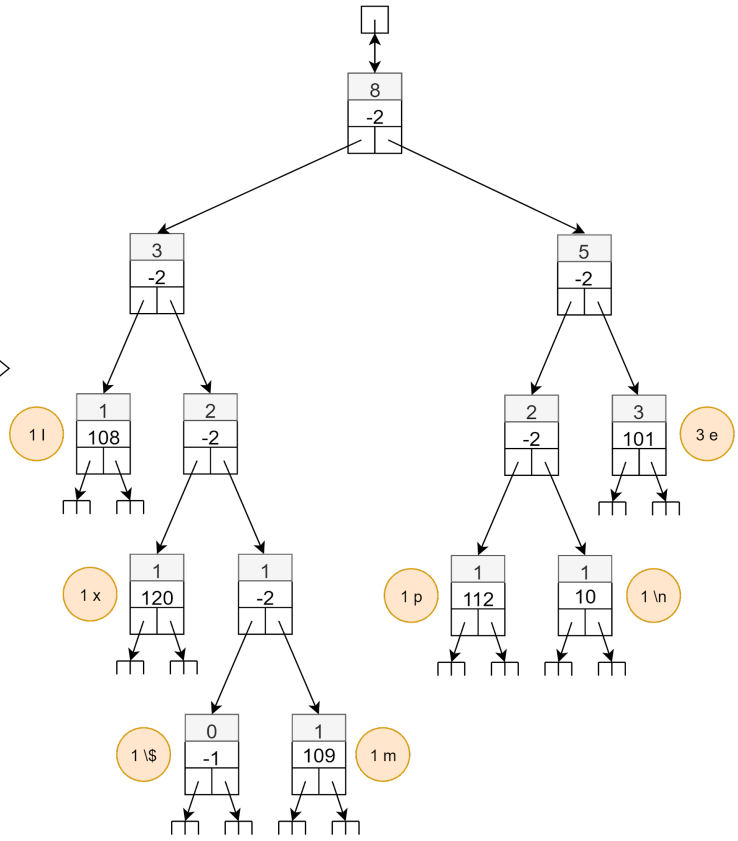
Exemple:



Compresser regroupe ensuite tous les arbres du tableau et extrait l'arbre final de type T_AB de façon à ce que les fréquences les plus faibles soient placées plus profondément dans l'arbre.

Exemple:

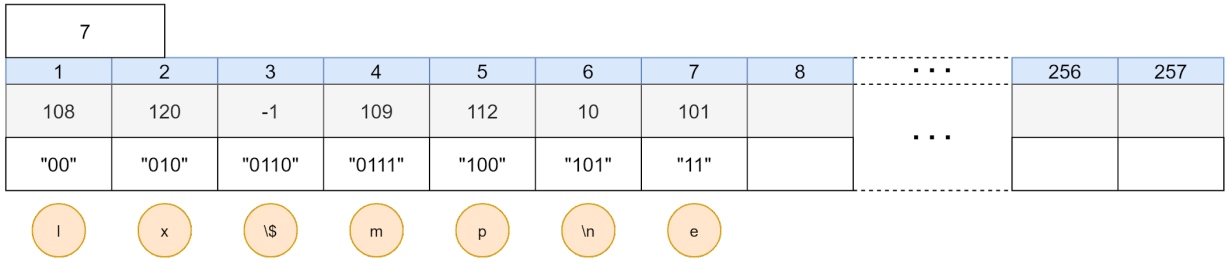
T_TAB_AB



A partir de ce T_AB, compresser crée la table de huffman de type T_TAB_HUFF, contenant les codes de huffman des caractères.

Exemple:

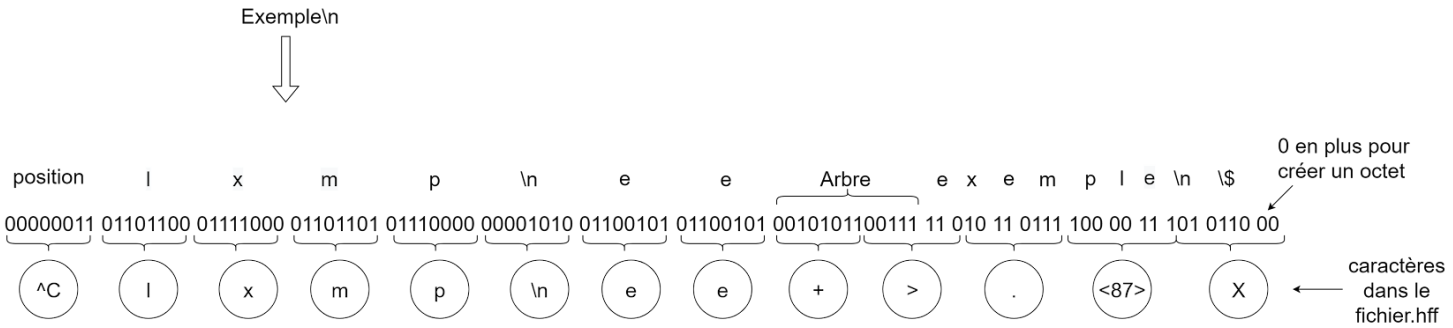
T_AB



Le programme crée ensuite le code de l'arbre qui est ici pour notre fil-rouge “0010101100111”, via une fonction du module AB. Il crée ensuite un fichier .hff dans lequel il insère la position du caractère \$

dans la liste des caractères, puis les autres caractères (le dernier caractère étant doublé), le code de l'arbre, le code du fichier et en dernier lieu le caractère de fin sous la forme d'octet.

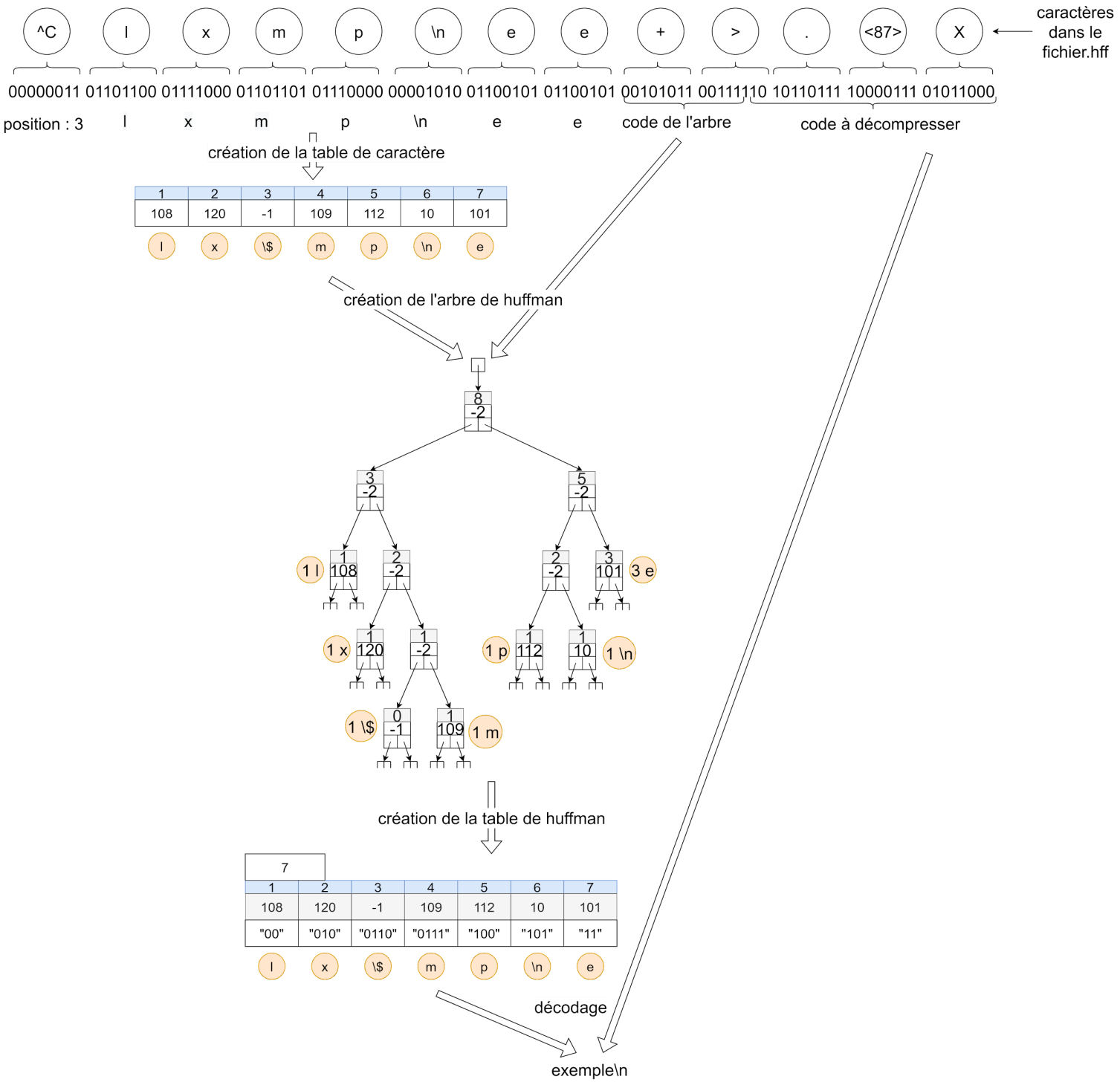
Exemple:



c. Présentation de l'algorithme de décompression

Décompresser ouvre tout d'abord le fichier .hff, stocke la position de \$ qui se trouve dans le premier octet lu et stocke la liste des caractères qui se trouve dans les octets suivants (la fin de la liste est donné car le dernier caractère est doublé) dans une table de caractère. Il récupère ensuite le code de l'arbre et recrée l'arbre de Huffman dans un T_AB grâce à ce code et à la liste de caractère qu'il a précédemment créé. Le programme recrée la table de Huffman dans un T_TAB_HUFF à partir de cet arbre. Enfin décompresser recrée le fichier décompressé en finissant de parcourir le fichier .hff et en insérant les caractères correspondants au code lu grâce à la table de Huffman jusqu'à arriver au caractère de fin de fichier \$.

Exemple:



1	2	3	4	5	6	7
108	120	-1	109	112	10	101
l	x	\$	m	p	\n	e

7						
1	2	3	4	5	6	7
108	120	-1	109	112	10	101
"00"	"010"	"0110"	"0111"	"100"	"101"	"11"
l	x	\$	m	p	\n	e

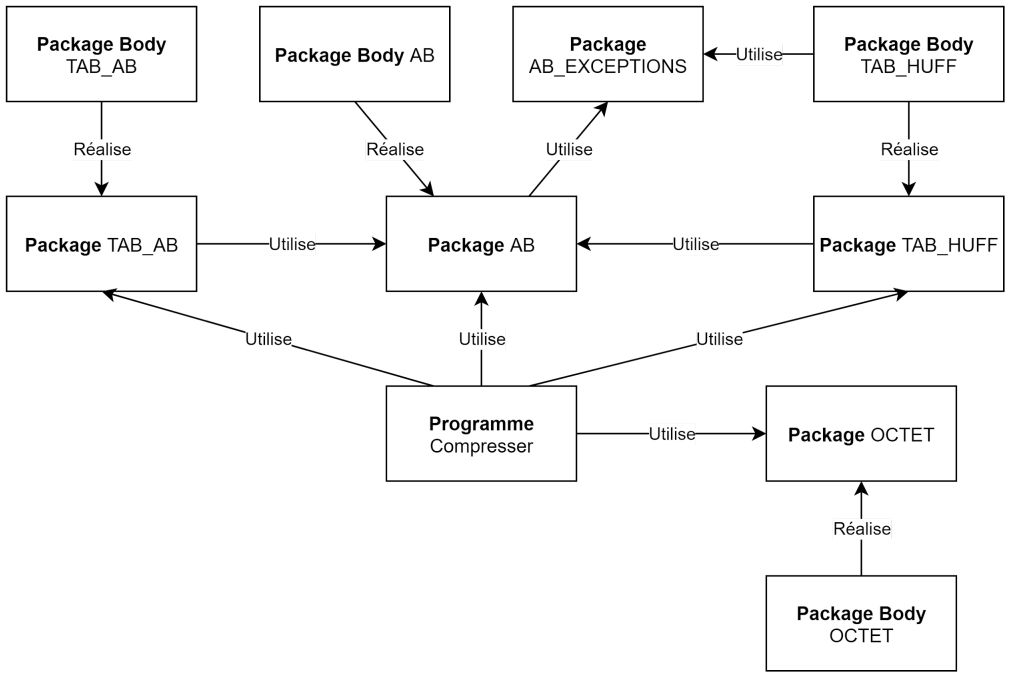
II. Choix réalisés

Un des premiers choix réalisés a été celui d'utiliser le type Unbounded_String pour les codes des caractères ou des arbres de Huffman, car ce type permet une manipulation plus simple et plus souple des chaînes de caractère par rapport au type String.

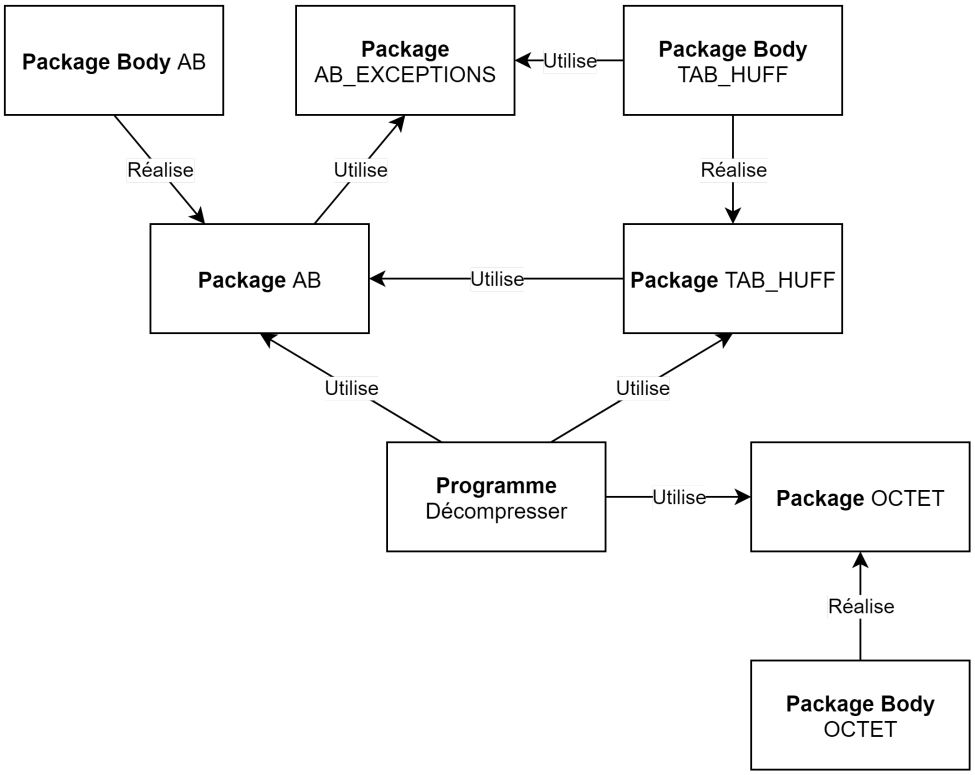
Nous avons ensuite choisi d'utiliser des sous-procédures au sein des procédures Afficher_AB et Creer_Table afin d'améliorer la clarté de ces dernières.

III. Architecture

Architecture du programme Compresser



Architecture du programme Decompresser



IV. Démarche de test

Pour tester si nos programmes fonctionnaient, nous avons tout d'abord créé des fichiers de tests pour chacun des modules réalisés, qui vérifiaient si chaque fonction et procédure que nous avions implémenté réalisait ce que nous voulions. Nous avons ensuite testé les programmes compresser et decompresser en créant des fichiers textes et en lançant les programmes via la ligne de commande. Nous avons pu ensuite vérifier qu'après compression et décompression, les fichiers retrouvés étaient bien identiques aux fichiers textes initiaux.

V. Difficultés et solutions

Le premier problème rencontré s'est posé durant l'étape de raffinage: il s'agissait de trouver un type permettant de stocker les caractères et la fréquence qui leur est associée de manière optimisée. Après avoir envisagé différentes possibilités telles que des tables de hachage ou un tableau dans lequel seraient stockés les couples les uns après les autres, nous avons décidé d'utiliser un tableau d'arbres unitaires (ne contenant qu'une feuille), cela simplifiant la tâche de créer l'arbre de Huffman.

Un nouveau problème a été découvert plus tard lorsque nous avons voulu enregistrer le caractère fin de fichier (/$). Au départ nous enregistrions des caractères dans nos différents types, et nous avions associé /$ au caractère codé par l'entier 3 de Latin1(correspond à fin de texte) : celui-ci marquait la fin du programme décompresser et si ce caractère était présent dans le fichier initial, la décompression s'arrêtait sans finir la décompression du fichier. Cette manière de le représenter induisait aussi que notre programme ne répondait pas au cahier des charges, puisque dans celui-ci il était indiqué que le caractère /$ était en plus et devait donc être différent des caractères de Latin1. Cela nous a conduit à modifier la manière dont sont enregistrés les caractères: nous avons décidé de les enregistrer directement par leur entier associé en Latin1, et pour le caractère fin de fichier par l'entier -1.

L'arbre de Huffman a lui aussi constitué une difficulté quant à la manière de le représenter: après quelques modifications il a finalement été choisi de l'associer à un arbre binaire dont les feuilles contiennent des couples fréquences/(entiers associés au caractère en Latin1), le caractère fin de fichier étant représenté par -1 et les noeuds n'étant pas des feuilles contenant -2.

Par ailleurs, nous souhaitions conserver certains types (T_AB et T_TAB_AB) définis dans nos modules en limited private, ce qui nous a conduit à effectuer de nombreux changements dans les fonctions des modules correspondants, notamment en créant des procédures de copie ou de tests d'égalité. Il a fallu aussi, pour la même raison, créer des procédures à la place de fonctions.

Des oublis de fonctions permettant de libérer la mémoire ont dû être comblés pour éliminer les erreurs liées à Valgrind.

VI. Organisation de l'équipe

Concernant la première partie du projet qui correspond à la conception des raffinages, nous avons fait le choix de réfléchir simultanément à chaque étape du raffinage, afin que notre vision de la résolution du problème soit la plus cohérente possible.

Pour la spécification des modules et l'écriture des tests des fonctions et procédures de ces derniers, nous avons cette fois divisé la tâche de sorte que l'un s'attache aux modules TAB_AB et TAB_HUFF et l'autre aux modules AB et OCTET.

La même division a été opérée pour l'implémentation de ces différents modules.

Quant à l'implémentation des programmes principaux Compresser et Decompresser, la tâche a encore une fois été divisée de sorte que chacun de ces deux programmes soit implémenté par une personne différente.

Pour ce qui est de la vérification et correction de nos programmes finaux, chacun de nous a vérifié les 2 programmes pour permettre une correction plus complète et sans oublis.

VII. Bilan technique

De notre point de vue, nos programmes répondent au cahier des charges.

Une perspective d'amélioration de nos programmes consisterait à pouvoir compresser des fichiers contenant des caractères codés dans un autre format que le format Latin1, comme par exemple le UTF-8. Il pourrait aussi compresser des fichiers contenant d'autres caractères que ceux des langues latines (ex: ß, ↔, 古, ᾏ, ش).

VIII. Bilan personnel et individuel

Nous avons au final passé au final environ 110h sur ce projet réparties comme suit:

~ 30h raffinages
~ 10h spécifications des modules
~ 10h implémentation des modules
~ 30h test et correction des modules
~ 5h implémentation des programmes
~ 10h test et correction des programmes
~ 10h rapport
~ 5h démonstration et présentation

Le temps de travail a été au final équitablement réparti entre les 2 membres de notre binôme.

Ce projet a été très enrichissant pour chacun d'entre nous. Il nous a permis en premier de nous améliorer dans le codage en langage ADA via la pratique, mais aussi de comprendre comment s'organiser quand on travaille à plusieurs sur un même projet. Il nous a aussi permis de comprendre et de mettre en pratique le cheminement que l'on doit avoir afin de passer d'un cahier des charges à un

programme fonctionnel. Nous avons aussi pu voir l'importance de la vérification de nos modules et programmes à chaque étape. Ces vérifications intermédiaires nous ont permis de gagner du temps au niveau de la vérification finale, puisque nous pouvions localiser plus précisément les erreurs commises et donc diminuer le temps de correction de tous nos programmes et modules.