,

Rapport Projet 3 — PIM

Auteurs :

VIGNAUX Adrien **BLANCHARD Enzo**

Professeur :

HAMROUNI Zouheir

Projet de première année de SN année : 2024-2025

Table des matières

1. Introduction

Lors de ce module de Programmation Impérative, nous avons déjà pu réaliser deux mini-projets individuels : le premier étant sur la création du jeu des 13 allumettes, le second étant sur l'application des modules et de la généricité en lien avec des LCA (Listes Chaînées Associatives).

1.1. Objectifs

Pour ce troisième et ultime projet du module, nous sommes en binôme, et il nous est demandé d'écrire deux programmes :

- Le premier doit compresser des fichiers textes en utilisant le codage de Huffman ;
- Le second doit permettre de les décompresser.

Ce rapport débutera sur une présentation large du problème pour se poursuivre par nos démarches de résolution. Nous présenterons ensuite nos différents choix ainsi que les principaux rendus de ce projet (modules par exemple). Puis, nous expliquerons notre gestion du travail en équipe, avec les diverses difficultés rencontrées et ce qui pourrait être amélioré à l'avenir. Nous clôturerons ce rapport par un bilan technique, puis personnel.

1.2. Description du problème

Le codage de Huffman permet de compresser, sans perte, des données telles que des textes, des images ou encore du son. Nous sommes ici intéressés par la compression de texte.

Ce principe revient à compter le nombre d'occurrences de chaque caractère présent dans un texte, puis de leur inscrire une série de bits dont la taille dépendra de la fréquence d'apparition du caractère dans le texte. Ainsi, un caractère "commun" sera codé sur peu de bits, tandis qu'un caractère "peu commun" sera codé sur davantage de bits.

Assurément, nous devrons permettre la compression ainsi que la décompression du texte, sinon il nous serait impossible de récupérer et utiliser son contenu initial, avant sa compression.

2. Contenu du projet

2.1. Structure générale du projet

Notre projet se regroupe en plusieurs fichiers qui seront explicités au fur et à mesure de ce rapport. Voici la liste :

- Les fichiers arbre.adb et arbre.ads : module "Arbre" permettant de manipuler l'Arbre de Huffman ;
- Les fichiers table.adb et table.ads : module "Table" permettant de manipuler diverses tableaux spécifiques tel que la Table de Fréquence et la Table de Huffman ;
- Le fichier compresser.adb : premier fichier principal réalisant la compression d'un texte à partir du principe de codage de Huffman ;
- Le fichier decompresser.adb : second fichier principal réalisant la décompression de notre texte préalablement compressé ;

2.2. Choix effectués

Pour aborder plus facilement notre travail, parlons de quelques choix essentiels que nous avons pris qui ont inévitablement influencé notre travail tout du long de ce projet.

Avant tout, il est important de noter que ces choix ne sont ni bons ni mauvais, ils se mêlent à un tout qui rend simplement notre travail différent des autres, tout comme ces derniers ont dû faire des choix qui les ont influencés dans leurs démarches de réflexion et de réalisation de ce projet.

Premièrement, nous avons décidé de privilégier majoritairement l'usage des tableaux plutôt que des Listes Chaînées Associatives (soit LCA) afin d'avoir un accès direct aux données, au détriment de ne pouvoir faire varier la capacité d'un tableau.

Cependant, cela est aussi lié à notre deuxième choix qui est d'utiliser un bloc "declare", qui nous permet de pallier ce problème. En effet, cela nous permet de définir une capacité qui dépend d'un calcul effectué par un sous-programme, afin de récupérer exactement un tableau de la taille souhaitée. En revanche, cela signifie aussi que tout ce qui est influencé par cette capacité doit être déclaré après cette dernière ; nous pouvons voir cela comme une déclaration en deux temps. C'est la solution que nous avons trouvé pour réaliser ce projet. Bien qu'elle puisse être différente de ce qui pouvait être initialement escompté, elle n'en reste pas moins valable, avec ses avantages et ses inconvénients. C'est pourquoi nous avons tenu à le préciser autant, pour que cela ne surprenne pas au premier abord. Bien entendu, nous avons ajouté des commentaires permettant de guider quiconque lira notre code.

Enfin, nous avons aussi décidé de rendre le module arbre public pour deux raisons. D'abord, cela nous facilitait la tâche lors des différentes manipulations et de la construction des sous-programmes (qui seront explicités ci-dessous). De plus, puisque ce module est exclusif à l'usage de notre compression (le module arbre pourrait contenir plus de manipulations, mais nous n'en avons pas besoin ici), alors personne ne pourra l'exploiter autrement que pour l'usage de la compression par codage de Huffman. Nous avons trouvé inutile de le rendre privé alors que ce n'est pas nécessaire. C'est donc une décision que nous avons prise afin de gagner en efficacité dans notre travail.

2.3. Architecture de l'application modules

Afin de structurer, de faciliter et d'épurer notre travail, nous devons mettre en place des modules en fonction des types de données utilisés et de comment les manipuler. Entrons dans le détail :

- Module Arbre :

  Qui dit arbre de Huffman dit qu'il nous faudra créer un nouveau type de données pour le représenter ; ce module est fait pour cela. En effet, l'arbre de Huffman est représenté par une LCA dans lequel on effectue un enregistrement de plusieurs éléments (que l'on appellera un nœud) :

  - un caractère ;
  - un entier (qui sera la fréquence d'apparition du caractère) ;
  - une branche gauche ;
  - une branche droite.

Les branches gauche et droite représentent un autre enregistrement, et ce, jusqu'à ce que les deux branches d'un nœud soient nulles. Il faut noter que, dans notre situation, il n'existe que deux situations.

Nous pouvons soit être sur une branche de l'arbre, donc il n'y a pas de caractère ou sa fréquence d'enregistrée, mais les branches gauches et droites sont bien affectées à un autre enregistrement. Sinon nous sommes sur une feuille, il n'y a donc pas d'autre branche associée, cependant on y retrouve un caractère ainsi que sa fréquence d'apparition dans le texte.

Ce nouveau type de données implique la création de fonctions et procédures permettant de le manipuler efficacement et rigoureusement. Voici une liste des actions possibles :

- Initialiser
- Detruire
- Enregistrer_Feuille
- Enregistrer_Branche
- Frequence_presente
- La_Frequence
- Taille
- Afficher_Feuille
- Afficher_Arbre

- Module Table :
  Nous allons manipuler plusieurs tableaux de tailles différentes, et contenant des éléments différents également. Pour ce faire, nous avons besoin de créer ce module de façon générique, afin de récupérer tous nos tableaux nécessaires, tout en ne se souciant pas davantage de la manipulation de ces derniers, puisque tout sera effectué ici, dans ce module.
  Nous utilisons au total cinq tableaux différents, qui ont assurément le même fonctionnement, mais absolument pas la même utilité ; d'où la nécessité de généricité ici. Afin de manipuler ces tableaux, voici les différentes actions possibles :
  - Initialiser
  - Taille
  - Enregistrer
  - Supprimer
  - La_Valeur
  - Afficher

- Module Conversion :
  Puisque nous avons fait le choix de travailler sur du texte plutôt que des octets, il est nécessaire de convertir les octets en entier ou en caractère et inversement dans plusieur programmes différents. Ainsi nous avons fait le choix de créer un module non générique qui regroupe ces fonctions, que voici :
  - Boolean_To_Octet
  - Integer_To_Octet
  - Caractere_to_Integer : convertit les caractères $'1'$ et $'0'$ en 1 et 0
  - Octet_To_Integer

2.4. Principaux algorithmes

Passons à présent aux différents sous-programmes principaux et essentiels au bon fonctionnement de la compression/décompression. Tout d'abord, retrouvons les algorithmes du fichier compresser.adb :

- Conversion_Texte :

La première étape avant de débuter une quelconque création d'un arbre ou d'une table de Huffman est de s'assurer que l'on récupère bien le texte à compresser, et que l'on puisse en lire son contenu. Nous avons donc mis en place ce sous-programme qui permet de récupérer, caractère par caractère, le texte initial et de l'insérer dans une variable en Unbounded_String. Ce dernier s'occupe assurément de la gestion des types ; la lecture s'effectuant sous forme d'octet, il est nécessaire de le convertir en caractère à concaténer à la suite du texte, jusqu'à ce qu'il soit complètement lu.

- Creer_Table_Frequence :

La compression par codage de Huffman nécessite la fréquence d'apparition des caractères en premier lieu, et pour les enregistrer nous allons créer un tableau de ces dernières. Pour ce faire, en sachant que les caractères usuels peuvent s'écrire en entier ne dépassant pas 256, nous avons utilisé les indices du tableau comme repère (afin d'enregistrer indirectement le caractère associé à la fréquence enregistrée).
En effet, prenons l'exemple du caractère 'a', valant 97, si il apparaît 13 fois dans le fichier que l'on souhaite compresser, alors la table de fréquence sera remplie tel que Table_freq[97] = 13.

- Arbre_Huffman :

La compression du texte demande le parcours de l'arbre de compression de Huffman. Pour cela nous allons créer un tableau d'arbrisseaux que nous allons dans un premier temps remplir de feuilles seulement.
Puis nous recherchons les deux arbrisseaux différent de fréquences minimum, et nous créons un noeud qui prendra les deux arbrisseaux en fils droit et gauche.
Nous plaçons ce nouvel arbrisseau à la place du premier minimum dans le tableau d'arbrisseaux et nous mettons à Zero la cellule du deuxième pour qu'elle ne soit plus prise en compte dans la recherche suivante de minimum.
Nous répétons ces opérations jusqu'à n'avoir qu'un seul arbre dans notre tableau, qui est notre arbre d'Huffman.

- Creer_Codage_Huffman :

Maintenant que notre arbre est créé, nous allons enregistrer leur code issu de ce dernier dans un tableau similaire à la table de fréquence, cependant nous n'allons pas enregistrer d'entiers pour les fréquences mais des Unbounded_String pour le code binaire du caractère (exemple : "10010").
Nous allons donc parcourir l'arbre jusqu'à tomber sur une feuille, puisque qui dit feuille dit code du caractère enregistré, ainsi que le caractère en question. C'est ainsi que notre tableau se remplit.

- Creer_Table_Huffman :

À présent, il nous faut enregistrer la signature de notre arbre pour que, lors de la décompression, nous puissions le reconstruire à l'aide de cette dernière.
Le fonctionnement est exactement le même que pour Creer_Codage_Huffman précédemment, cependant nous enregistrons cette fois-ci le caractère ainsi que sa position dans l'arbre pour que, lors de la reconstruction, nous n'ayons qu'à lire dans l'ordre les caractères de cette table.
Il est important de noter que le premier élément de ce tableau est la position du caractère de fin ; puisqu'il n'a pas de valeur représentative (on ne pourra pas écrire -1 dans le fichier), nous optons pour cette solution, étant celle mentionnée dans le sujet.

- Compresser_Table_Huffman :

Maintenant, tous les éléments sont en place, nous en sommes rendus à la dernière étape et la plus concrète de toutes : la compression du fichier. Tout d'abord, il nous faut écrire dans le fichier compressé notre table de Huffman, qui sera la première à être décodée pour la décompression. Cela se contente simplement d'effectuer une boucle qui écrit chaque élément de la table un par un, sous forme d'octet bien sûr.
Notons deux points importants : d'abord, puisque la conversion d'un entier en octet se fait sans embûche pour la reconnaissance des caractères, nous n'avons aucun besoin d'écrire bit par bit ce dernier jusqu'à pouvoir écrire l'octet sur le fichier (rappelons que l'on ne peut qu'écrire octet par octet). De plus, pour savoir quand la lecture s'interrompt, c'est-à-dire quand est-ce que la table de Huffman est décodée, la solution proposée est d'écrire deux fois le dernier caractère ; cette double écriture est essentielle pour ne pas lire trop d'octets en pensant que c'est toujours la table de Huffman, alors que non.

- Compresser_Huffman :

En second temps, nous devons écrire la signature de l'arbre de Huffman, afin que lors de la décompression, la reconstruction de l'arbre soit possible.
La méthode adoptée ici est d'écrire un '0' lorsque l'on parcourt l'arbre vers la gauche, puis lorsqu'on atteint une feuille, on écrit un '1'. Le sous-programme s'occupe donc de cette mission, en sachant que cette fois-ci, nous devons écrire bit par bit puisqu'il est tout à fait plausible que la signature de l'arbre ne se termine pas par un octet plein (ce sera même peu commun).
La gestion de cette écriture ne sera pas explicitée ici puisque nous ne présentons que les principaux algorithmes du programme, en revanche nous avons déjà abordé le sujet lors du chapître "Choix effectués" précédemment.

- Compresser_Texte :

Finalement, le dernier sous-programme principal de cette partie de compression est celui de la compression du texte. À l'aide du texte récupéré au tout début du programme, nous allons le parcourir caractère par caractère, puis écrire le code correspondant à ce dernier, enregistré dans la table créée par l'algorithme Creer_Codage_Huffman (voir ci-dessus).
La seule subtilité ici, c'est le caractère de fin de fichier : lorsque tout le texte aura été parcouru, alors c'est à ce moment que ce caractère sera écrit, bien entendu avec son code issu de l'arbre de Huffman.

Passons maintenant aux sous-programmes du fichier decompresser.adb :

- Decoder_Table_Huffman :

Les premiers octets du fichier compressé représentent les éléments contenus dans la table de Huffman. Nous avons donc besoin d'un sous-programme qui recrée cette dernière en décodant les octets concernés. Rappelons que, pour indiquer la fin de lecture des octets de la table de Huffman, nous avons doublé l'écriture du dernier caractère. Nous enregistrons donc l'octet précédent que nous comparons à l'octet précédent, puis nous effectuons la conversion de l'octet courant en caractère, s'inscrivant dans la table dans l'ordre conventionnel, jusqu'à ce que ces deux octets soient similaires.

- Reconstruire_Arbre_Huffman :
  À présent, nous allons pouvoir recréer l'arbre de Huffman puisque nous avons toutes les informations permettant de le remplir. Pour ce faire, nous utilisons les bits représentant la signature de l'arbre dans le fichier compressé, tel que si le bit courant est un '0', nous poursuivons la construction de l'arbre à gauche, sinon c'est un '1', ce qui signifie que nous sommes arrivés sur une feuille. À ce niveau, deux choix s'offrent à nous :
  - Nous sommes encore du côté gauche de l'arbre, donc nous avons une feuille à gauche que nous remplissons à l'aide la table de Huffman, puis nous passons au côté droit ;
  - Nous sommes déjà du côté droit de l'arbre, donc nous avons une feuille à droite et nous retournons à la branche supérieure pour poursuivre son côté droit.

- Decompresser_Texte :

Finalement, il ne nous reste plus qu'à décompresser le texte et d'écrire les caractères correspondants dans le fichier décompressé. Pour ce faire, nous allons lire bit par bit le reste des octets du fichier compressé pour nous permettre de parcourir l'arbre ('0' : on passe par la branche gauche, '1' : on passe par la branche droite). Au préalable, on vérifie si le nœud où nous sommes arrivés est une feuille, auquel cas cela signifie que nous avons trouvé le caractère à écrire. Il ne manque plus qu'à récupérer ses informations, et de l'écrire dans le fichier décompressé. L'algorithme s'arrête lorsque le caractère de fin de fichier est trouvé.

3. Gestion du travail

3.1. Organisation de l'équipe

La répartition des tâches lors de la réalisation de ce projet s'est effectué comme telle :

Modules	Sous-programmes	Spécification		Codage		Test		Relecture	
		Adrien	Enzo	Adrien	Enzo	Adrien	Enzo	Adrien	Enzo
Arbre	Initialiser		X	X			X	X	
	Détruire	X		X			X	X	
	La Fréquence		X		X		X	X	
	Enregistrer Feuille	X		X			X	X	
	Enregistrer Branche	X		X			X	X	
	Fréquence présente		X		X		X	X	
	Taille		X	X			X	X	
	Afficher Caractère	X		X			X	X	
	Afficher Arbre	X		X			X	X	
Table	Initialiser		X		X	X			X
	Taille		X		X	X			X
	Enregistrer		X	X		X			X
	Supprimer	X		X		X			X
	La Valeur	X		X		X			X
	Afficher		X		X	X			X
Compresser	Créer Table Fréquence		X		X	X			X
	Octet To Integer	X		X		X			X
	Conversion Texte	X		X		X			X
	Tri Table Fréquence	X		X		X			X
	Boolean To Octet		X		X	X			X
	Integer To Octet	X		X		X			X
	Ecrire Octet		X		X	X			X
	Arbre Huffman	X		X		X			X
	Créer Codage Huffman	X		X		X			X
	Créer Table Huffman	X		X		X			X
	Compresser Huffman		X		X	X			X
	Caractère to Integer		X		X	X			X
	Compresser Texte	X		X		X			X
Décompresser	Integer To Octet	X		X		X		X	
	Octet To Integer	X		X		X		X	
	Décoder Table Huffman		X		X		X	X	
	Lire Bit		X		X		X	X	
	Reconstruire Arbre Huffman		X		X		X	X	
	Décompresser Texte		X		X		X	X	
Conversion	Boolean To Octet		X		X		X	X	
	Integer To Octet		X		X		X	X	
	Caractere to Integer	X		X			X	X	
	Octet To Integer	X		X			X	X	

3.2. Difficultés rencontrées

Durant la réalisation de ce projet, nous avons pu rencontrer certaines difficultés qui valent la peine d'être explicitées.

Premièrement, nous avons eu du mal à comprendre le fonctionnement de l'arbre de Huffman, ou plutôt comment elle s'implémente d'un point de vue algorithmique. Cela nous a demandé un peu plus de temps que prévu lors des prémices du projet, afin de partir sur de bonnes bases.

De plus, nous avions toujours en tête d'optimiser le programme dès que possible, cependant pour un projet qui commence à avoir une plus grande ampleur il est essentiel de débuter sur une base fonctionnelle (même si non optimisée), puis par la suite de l'améliorer du mieux que possible. Ainsi, nous partions souvent dans des idées bien trop complexes, qui n'aboutissaient pas forcément.

Enfin, il nous a été primordial de bien gérer notre temps, afin de pouvoir avancer efficacement dans la complétion du projet et de rendre notre travail dans les temps lorsque c'était demandé. C'était une difficulté continue qui ne devait être négligée à aucun moment, sous peine de retour de bâton imprévu.

4. Conclusion

4.1. Bilan technique

Ce projet nous a permis de produire pour la première fois un résultat concret à l'aide de la programmation : dans notre situation, nous avons réussi à faire fonctionner les deux programmes, que ce soit pour compresser ou décompresser un fichier texte. Nous avons aussi pu réfléchir par nous-même afin de proposer une solution qui nous soit propre.

Cependant, il est important de noter que nous avons plusieurs idées pour améliorer davantage notre projet. D'abord, nous aurions aimé pouvoir gérer une plus grande quantité de fichiers différents, sans que cela ne vienne poser souci, ou encore pouvoir compresser / décompresser plusieurs fichiers simultanément ; c'est-à-dire nous aurions aimé traiter davantage les cas limites de notre projet. Nous avons préféré privilégier un fonctionnement basique mais tout à fait conforme plutôt que plusieurs fragments de projets à peu près fonctionnels.

De plus, il aurait été intéressant d'automatiser davantage les démarches de tests pour une justification plus claire et concise du bon fonctionnement de nos programmes. Deux fichiers de programme de tests (pour compresser et décompresser) auraient été une bonne solution.

4.2. Bilan collectif

Nous avons passé environ 6 heures sur la conception des programmes pour chaque personne du binôme, de même pour la mise au point. Nous avons passé environ 14 heures chacun sur l'implémentation et 5 heures sur le rapport. Ce qui au final fait en combiné environ 50 heures.

4.3. Bilan personnel et individuel

4.3.1. De Adrien VIGNAUX

Ce projet est très enrichissant au niveau du travail en équipe car c'est le premier projet long que l'on fait à deux.

4.3.2. De Enzo BLANCHARD

Finalement, ce projet était notre première approche du monde de l'ingénierie : entre la gestion d'équipe et du travail, les deadlines à respecter ou encore le processus de création, d'innovation pour un rendu encore et toujours plus performant, efficace, ce projet était assez complet et nous a permis de pleinement manipuler tous les aspects de la programmation impérative.