
# Rapport du projet de Programmation Impérative

# Codage de Huffman

Xu Thierry

Bonetto Tom

Équipe IJ09

Département Sciences du Numérique - Première année

2021-2022

## Table des matières

# 1. Objectif rapport

Ce rapport a pour but de faciliter la compréhension de notre version du codage de Huffman mais également de justifier nos choix dans la manière de coder cet algorithme (modules, structures de données, programmes test). L'autre objectif est de faire un compte rendu sur notre méthode de travail et notre organisation en tant qu'équipe mais aussi sur les difficultés rencontrées et les solutions apportées ainsi que sur notre bilan personnel dans ce projet et les possibles améliorations.

# 2. Introduction

Le codage de Huffman est un codage utilisé pour la compression sans perte de données telles que les textes. Son principe est de définir un nouveau codage des caractères, codage à taille variable qui tient compte de la fréquence (le nombre d'occurrences) des caractères dans le texte : les caractères dont la fréquence est élevée seront codés sur moins de bits et ceux dont la fréquence est faible sur plus de bits.

L'objectif de ce projet est d'écrire deux programmes, le premier qui compresse des fichiers en utilisant le codage de Huffman et le second qui les décompresse.

# 3. Architecture logicielle et structures de données

## 3.1 Architecture logicielle

Nous avons créé 6 modules différents en complément de ceux de Ada pour pouvoir réaliser le code :

1. arbre : définit la structure d'un arbre binaire et les différentes fonctions et procédures qui permettent de la manipuler, le modifier et l'afficher.

2. pile : définit la structure d'une pile et les différentes fonctions et procédures qui permettent de la manipuler, le modifier et l'afficher.

3. lca : définit la structure d'une liste chaînée associative et les différentes fonctions et procédures qui permettent de la manipuler, le modifier et l'afficher.

4. tableau : définit les 3 types tableaux ainsi que leur procédure d'affichage.

5. openfile : comprend les différentes procédures concernant la conversion binaire à octet et inversement, ainsi que la lecture et l'écriture dans des fichiers.

6. compression : contient les procédures et fonctions permettant de réaliser toutes les étapes pour compresser (arbre de huffman,  table de huffman, liste octet).

7. decompression : contient les procédures et fonctions permettant de réaliser toutes les étapes pour décompresser (reconstruire arbre de Huffman,  reconstruire liste octet).

Nous utilisons également certains modules prédéfinis de Ada :

1. Ada.Text_I0

2. Ada.Integer_Text_I0

3. Ada.Streams.Streams_I0 : pour la lecture et l'écriture dans les fichiers

## 3.2 Structures de données

1. Arbre binaire

```
private

    type T_Noeud;  -- Noeud d'un arbre

    type T_Arbre is access T_Noeud; -- Le pointeur sur T_Noeud

    type T_Noeud is record
        Frequence: Integer; -- Fréquence de l'arbre
        Gauche: T_Arbre;    -- Sous arbre gauche
        Droite: T_Arbre;    -- Sous arbre droit
    end record;
```

L'arbre binaire est créé puisque tout le principe du codage de Huffman repose sur cette structure.

2. Liste chaînée associative

```
private

     type T_Cellule; -- Cellule de la liste

     type T_LCA is access T_Cellule;   -- Le pointeur sur T_Cellule

     type T_Cellule is
          record
               Cle : Integer;      -- Une clé
               Donnee : T_Donnee;  -- L'élément générique contenu dans la cellule
               Suivant : T_LCA;    -- La cellule suivante
          end record;

end LCA_projet;
```

Nous avons eu besoin d'une liste chaînée associative pour pouvoir stocker les différentes feuilles afin de construire l'arbre de Huffman, cette même structure sera utilisée pour stocker les nœuds intermédiaires lors de cette construction.

3. Pile

```
private

    type T_Tab_Elements is array (1..Capacite) of T_Element;

    type T_Pile is
        record
            Elements : T_Tab_Elements;  -- les éléments de la pile
            Taille: Integer;        -- Nombre d'éléments dans la pile
        end record;

end Piles;
```

4. Tableaux

```
package Pile_Caractere is
        new Piles (Capacite => 8, T_Element => Character);
use Pile_Caractere;

type T_Tab_character is array(1..5000) of character;

type T_Tab_symbole is record
    Elements : T_Tab_character; -- les éléments du tableau
    Nb_Elements : Integer;      -- Nombre d'éléments dans le tableau
end record;

type T_Tab_integer is array (1..5000) of Integer;

type T_Tab is record
    Elements : T_Tab_integer;  -- les éléments du tableau
    Nb_Elements : Integer;     -- Nombre d'éléments dans le tableau
end record;

type T_char is
    record
        char: Integer;
        binaire: T_Pile;
    end record;

type T_Tab_Huff_char is array(1..256) of T_char;

type T_Tab_Huff is record
    Elements : T_Tab_Huff_char;  -- les éléments du tableau (T_char)
    Nb_Elements : Integer;       -- Nombre d'éléments dans le tableau
end record;
```

Pour stocker nos différentes données nous avons eu besoin de 3 types de tableau différents : un tableau qui stocke les caractères (T_Tab_symbole : pour stocker le texte à compresser par exemple), un qui stock des entiers (T_Tab : pour stocker la liste des octets du fichier compressé lors de la décompression) et enfin un tableau qui stocke des entiers avec une pile (T_Tab_Huff_char : Pour stocker la représentation binaire des différents symboles du texte avec leur représentation binaire stocker dans la pile de caractère).

# 4. Principaux algorithmes et tests des programmes

## 4.1 Principaux algorithmes

1. Construire_Arbre_Huff : fonction qui renvoie l'arbre de Huffman sous la forme d'un arbre binaire de type T_Arbre, l'arbre est construit à partir du tableau de fréquence des différents symboles du texte (Tab_Freq : in Type T_Tab).

```
Arbre de Huffman :
(42)
\--1--(25)
|       \--1--(15)
|       \--0--(10)
|       |       \--1--(5)
|       |       |       \--1--(3)
|       |       |       \--0--(2)
|       |       |       |       \--1--(1)
|       |       |       |       |       \--1--(1)
|       |       |       |       |       \--0--(0)
|       |       |       |       \--0--(1)
|       |       \--0--(5)
\--0--(17)
|       \--1--(9)
|       |       \--1--(5)
|       |       \--0--(4)
|       |       |       \--1--(2)
|       |       |       \--0--(2)
|       \--0--(8)
|       |       \--1--(4)
|       |       \--0--(4)
```

2. Construire_Liste_binaire : procédure qui crée la suite binaire représentant l'arbre par parcours infixe, suite binaire stockée dans un tableau de type T_Tab_symbole et obtenu à partir de l'arbre de Huffman.

```
Liste binaire :
[ 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1 ]
```

3. Construire_Tab_Huff : fonction qui renvoie la table de Huffman sous la forme d'un tableau de type T_Tab_Huff, table créée à partir de la représentation de l'arbre par parcours infixe.

```
Table de Huffman :
[ 'm'  --> [ 0, 0, 0 ]
  'x'  --> [ 0, 0, 1 ]
  '\n' --> [ 0, 1, 0, 0 ]
  'l'  --> [ 0, 1, 0, 1 ]
  ' '  --> [ 0, 1, 1 ]
  't'  --> [ 1, 0, 0 ]
  ':'  --> [ 1, 0, 1, 0, 0 ]
  '\$' --> [ 1, 0, 1, 0, 1, 0 ]
  'd'  --> [ 1, 0, 1, 0, 1, 1 ]
  'p'  --> [ 1, 0, 1, 1 ]
  'e'  --> [ 1, 1 ] ]
```

4. Construire_Liste_octet : procédure qui permet de créer un tableau de type T_Tab comprenant la liste des symboles du texte en tant qu'octet dans l'ordre du parcours infixe de l'arbre. Permet également de mettre la position du caractère de fin de texte '\$' en tête de tableau ainsi que de dédoubler le dernier élément du tableau pour indiquer sa fin, indication qui sera utile pour la décompression.

```
[ 8, 109, 120, 10, 108, 32, 116, 58, 100, 112, 101, 101 ]
```

5. Stockage : fonction qui permet de lire le fichier à compresser et de stocker le texte contenu dans ce fichier dans un tableau de type T_Tab_symbole.

6. Écriture : procédure qui permet de mettre en forme les différents éléments à insérer dans le fichier compressé (Texte compressé, représentation de l'arbre en binaire par parcours infixe, liste des symboles en tant qu'octet), informations misent sous forme binaire pour ensuite créer des octets, octets qui constitueront le fichier compressé.

```
[ 8, 109, 120, 10, 108, 32, 116, 58, 100, 112, 101, 101, 25, 201, 124, 226, 215, 117, 238, 102, 110,
137, 156, 92, 220, 197, 243, 107, 156, 104, 149 ]
```

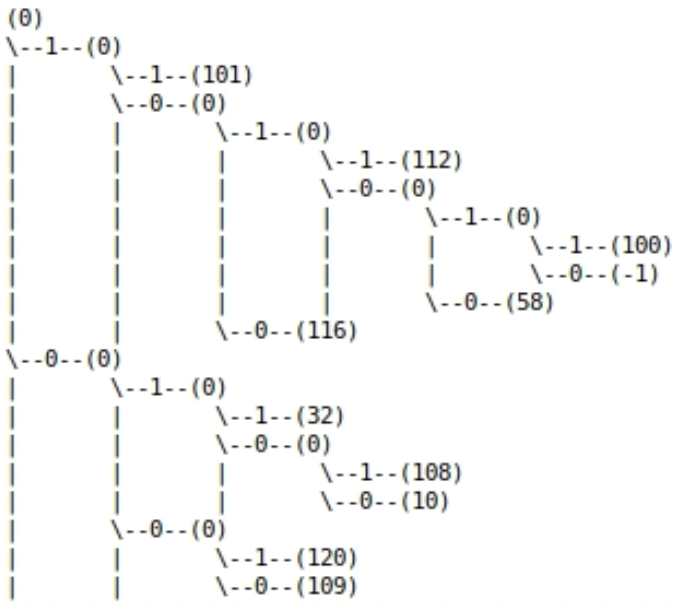
7. Lecture : procédure qui permet de lire un fichier compressé (composé d'octet) et de traduire ses octets en informations binaires pour par la suite identifier le texte compressé, la représentation de l'arbre en binaire par parcours infixe et enfin la liste des symboles en tant qu'octet et les stocker dans les structures de données adaptées. Après lecture on obtient nos 3 informations utiles à la décompression.

```
[ 8, 109, 120, 10, 108, 32, 116, 58, 100, 112, 101 ]

[ 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1 ]

[ 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1,
0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1,
1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1 ]
```

8. Reconstruire_Liste_octet : procédure qui permet de reconstruire la liste d'octets avant le dédoublement du dernier élément et qui remet la représentation du '\$' en tant qu'octet (-1) au bon endroit.

```
[ 109, 120, 10, 108, 32, 116, 58, -1, 100, 112, 101 ]
```

9. Reconstruire_Arbre_Huff : procédure qui permet de reconstruire l'arbre de Huffman à partir de la table de Huffman.




### 4.2 Tests des programmes

Nos structures de données (Arbre, Lca, Pile) possèdent toutes des programmes de tests (test_arbre, test_pile, test_lca) qui ont étés créés avant même de s'attaquer au codage de la compression et de la décompression, ces programmes de tests permettent de vérifier le bon fonctionnement de toutes les procédures et fonctions des modules permettant la manipulation de ces structures. On crée chacune de ces structures de données en taille réduite et on applique les différentes manipulations en vérifiant qu'elles ont étées appliquées correctement. Le module tableau ne fait pas l'objet d'un module de test puisque celui-ci contient seulement des procédures d'affichage, de plus nous utilisons les tableaux comme structure de stockage et aucune manipulation particulière ne sont effectuées dessus.
Pour ce qui est des tests de la compression et la décompression, il n'est pas réellement possible de faire des programmes pour les tester car le résultat à analyser est plutôt complexe, la seule manière de vérifier leur bon fonctionnement est d'afficher au fur et à mesure les différentes données. Nous avons créé deux procédures test_compression et test_décompression qui effectuent les différentes actions de la compression et de la décompression du texte pris en exemple dans le sujet et nous avons vérifié à chaque étape si le résultat correspondait à celui du sujet.

## 5. Difficultés rencontrées et solutions adoptées

La première difficulté que nous avons rencontré concerne la création de l'arbre de Huffman, celui-ci est créé à partir de la liste des fréquences des différents symboles que comporte le texte. Pour la liste de fréquence nous avions d'abord opté pour un tableau de T_Element, où T_Element serait un enregistrement comportant le symbole et sa fréquence associée. Finalement nous avons opté pour un simple tableau d'entiers ou la fréquence d'un symbole est rangée à l'indice correspondant au code

ASCII du symbole (exemple : si ‘X’ apparaît 4 fois dans le texte alors on range 4 à la case du tableau qui a pour indice 88 car le code ASCII de 88 est ‘X’), nous avons fait ce choix car les textes à compresser sont à priori uniquement composés de symbole de la table ASCII.

Une fois la table de fréquence créée se pose la question de la construction de l’arbre. Les différentes feuilles qui vont composer l’arbre demandent une structure de données capable de les stocker et ceci de manière triée. Nous avions d’abord pensé à un tableau mais nous avons rapidement opté pour une lca avec une procédure qui enregistre avec une relation d’ordre pour trier automatiquement tout élément qui serait ajouté à la lca. L’avantage de cette proposition est d'éviter l’utilisation d’algorithmes de tri pour les tableaux, algorithmes qui peuvent rapidement devenir coûteux si les tailles en jeu sont importantes, de plus lors du processus de création de l’arbre nous sommes amenés à supprimer les deux premiers éléments de la lca, chose plus facile à faire que si c’était un tableau.

Une difficulté mineure rencontrée est liée à l’élément ‘\$’ qui est considéré comme une chaîne de caractères et non un simple caractère par Ada, il ne fait de plus pas partie de la table ASCII ce qui pose problème lors de sa conversation en tant qu’octet, nous avons donc décidé d’utiliser -1 pour sa valeur en octet et de ne jamais manipuler ‘\$’ directement à part dans l’affichage.

Une autre difficulté rencontrée vient du stockage des représentations binaires des différents symboles. Nous aurions pu opter pour un tableau mais nous avons choisi une pile de caractères (on empile les différents bits un à un). Lors de la décompression pour recréer l’arbre de Huffman, nous avons besoin de parcourir bits par bits le code binaire des symboles, utilisé une pile avec une procédure qui dépile permet de simplifier cette opération.

# 6. Organisation de l’équipe et bilan technique

## 6.1 Organisation de l’équipe

Globalement, la réflexion sur le projet s’est toujours faite à deux, il n’y a pas de parties faites de manière complètement personnelle par un des membres du groupe. Nous avons tout de même été obligés de répartir le travail notamment pour respecter les échéances. Le raffinage de la décompression a été principalement réalisé par Thierry pendant que je codais les modules des structures de données et leur programme test associé. Pour ce qui est de la compression, Thierry a réalisé en grande partie les fonctions et procédures utiles à la construction de l’arbre et de la table de Huffman pendant que je m’attelais à l’ouverture et l’écriture de fichier. Pour la décompression nous avons suivi le même schéma, je me suis intéressé à la lecture de fichier pendant que Thierry faisait la reconstruction de l’arbre et de la table de Huffman. Les interfaces des modules et les procédures de tests pour compression et décompression ont été réalisées ensemble. Enfin la correction du raffinage a été faite par Thierry pendant que je rédigeais le rapport et le manuel utilisateur.

## 6.2 Bilan technique

### 6.2.1 État de l'avancement

Tous les modules sont terminés, la compression fonctionne (en tout cas pour l'exemple du sujet), la décompression fonctionne également. Manque de temps pour s'occuper des erreurs de valgrind et des exceptions.

### 6.2.2 Perspectives d'améliorations

Dans ce projet, l'utilisateur n'a quasiment pas accès aux programmes, il se contente de donner un fichier contenant un texte qu'il souhaite compresser et exécuter le programme principal. Il y donc très peu d'exceptions à gérer quant à l'utilisation du programme, cependant nos codes semblent peu robustes puisque, nous n'avons quasiment pas de bloc gérant les exceptions car celles-ci sont très compliquées à estimer, de plus il y a sûrement des problèmes liés à la mémoire dynamique (valgrind). En matière d'évolution, le codage de Huffman ne sert pas seulement à compresser des textes mais également des images et des sons, il est donc possible d'améliorer notre code pour que celui-ci gère également la compression de fichiers JPEG et de fichiers MP3.

# 7. Bilans personnels

## 7.1 Bilan de Xu Thierry

Bien que toutes les notions de modules, généricité, types abstraits de données ont été abordées durant les mini projets et les tps, ce projet m'a permis de renforcer ces acquis. M'étant occupé principalement des fonctions et procédures pour la compression et la décompression, j'ai rencontré un gros soucis avec les pointeurs et la récursivité dans la reconstruction de l'arbre lors de la décompression. Être rigoureux et réfléchir étape par étape m'a permis d'identifier la source du problème et adapter l'algorithme en conséquence. Ainsi, je me suis rendu compte que sur un projet de cette envergure, il est indispensable d'effectuer des tests au fur et à mesure même au sein d'une même fonction pour suivre l'avancement et identifier les problèmes. Comme Tom, c'est la première fois que je passe autant de temps sur un travail de programmation. Personnellement, je ne m'attendais pas à terminer la décompression du fichier avant l'échéance. Il est cependant agréable de mener à bien un projet de cette ampleur qui m'a apporté toute la rigueur qui me manquait.

## 7.2 Bilan de Bonetto Tom

Avec les travaux précédents en Ada (mini projet et tps), j'avais déjà assimilé en grande partie les notions de modules, généricité, types abstraits de données,  mais je n'avais encore jamais appliqué réellement toutes ces notions en même temps, dans un travail de cette envergure.

C’est pour ma part la première fois que je passe autant de temps sur un travail de programmation, je n’ai pas l’habitude de coder dans mes heures libres et je me rends compte à quel point il est difficile de mener un projet de ce genre à son terme. Sur le papier, le sujet de ce projet ne semble pas très difficile, mais on se rend compte très rapidement que beaucoup de problèmes surviennent. On a, je pense, passé plus de temps sur la correction de nos programmes que sur la conception, sûrement dû au fait que ce projet faisait appel à beaucoup de fonctions récursives. C’est en fait mon premier gros projet d'informatique, ce qui m'a apporté toute l’expérience qui va avec la conception d’un projet de ce genre.

## 8. Indication

1. Comment représenter l’arbre de Huffman ?

Avec un type abstrait de données tel qu’un arbre binaire parfait, c'est-à-dire que chaque nœud possède un sous arbre droit et gauche et une donnée qui indique la fréquence du nœud (fréquence qui correspond à la somme des fréquence des sous-arbre de ce nœud).

2. Est-ce que le codage d’un caractère est toujours composé d’au plus 8 bits ?

Non, dans le cas où il y a beaucoup de caractères différents dans le texte à compresser, il se peut que l’arbre de Huffman soit de taille très grande et que certain code de caractère dépasse ainsi l’octet.

3. Comment représenter le code d’un caractère ?

Le code d’un caractère est une suite binaire qui suit donc un certain ordre, on peut représenter ce code avec un type abstrait de données tel que la pile, on empile simplement les bits un à un et on dépile les éléments pour obtenir le code du caractère.

4. Comment représenter la table de Huffman ?

Avec un tableau de 256 éléments, éléments de type enregistrement qui permet d’avoir un caractère et son code associé stocké dans une pile.

5. La position du symbole de fin de fichier peut-elle toujours être représentée sur un octet ?

Si les caractères présents dans le texte compressé sont uniquement des caractères de la table ASCII, il est normalement impossible que dans l’arbre il y ait 256 feuilles avant celle du symbole de fin de fichier, ce qui veut dire que la position du symbole sera toujours inférieure à 256 et donc représentable sur un octet.

6. Lors de la décompression, comment reconstruire l’arbre de Huffman ?

En suivant les bits des codes des caractères on recrée chaque branche de l’arbre par récursivité. On ajoute à la fin de chaque branche c’est-à-dire sur chaque feuille le code ASCII du symbole, ce qui sera utile pour décoder les symboles du texte d’origine lors de la décompression.

7. Lors de la décompression, comment faire pour décoder les symboles du texte d’origine ?

En suivant les bits du texte codé en binaire et en parcourant l’arbre, on identifie chaque symbole du texte d’origine.