Département Sciences du Numérique

Programmation Impérative

Projet : Codage de Huffman

Rapport

Aicha Bennaghmouch

Nathan Foucher

Groupe CD07

Résumé

Ce rapport a pour but de rappeler les éléments du cahier des charges du projet et d'énoncer les différentes solutions retenues pour y répondre. Le présent document aura aussi pour objectif d'expliciter la structure et l'architecture de l'application, les choix majeurs de fonctionnement réalisés et la répartition du travail en équipe. Enfin nous énoncerons ici aussi les différents choix effectués par notre équipe face aux problèmes ayant été rencontrés.

Table des Matières

Introduction

Les masses de données informatisées étant de plus en plus importantes de nos jours, il est devenu quasi essentiel voir essentiel de limiter les demandes en capacité de stockage de ces derniers. L'une des solutions est d'utiliser des méthodes de compression (ici sans pertes) pour réduire le poids des fichiers. Une des méthodes pour réaliser cette compression est d'utiliser un arbre de Huffman. Ce dernier consiste à recoder les éléments stockés dans un fichier selon leurs nombres d'occurrences. On se concentrera pour ce projet sur la compression et la décompression de fichiers textes. Ainsi pour un caractère habituellement codé sur 8 bits, plus l'un d'eux sera présent dans un texte plus on le codera sur moins de bits afin de réduire la taille globale d'un fichier.

I. Début du projet

1. Appréhension de la structure globale du sujet

Le sujet à pour objectif final de créer deux programmes, compresser et décompresser, utilisant l'arbre de Huffman introduit précédemment. On comprend vite que tout le projet va se concentrer sur la création et l'utilisation d'un arbre de Huffman, le codage des interfaces utilisateurs n'est qu'une partie minime voire anecdotique de ce dernier. On remarque aussi que le premier problème que nous allons rencontrer sera de travailler avec des fichiers considérés comme binaires. En effet le codage d'un fichier texte pouvant varier (UTF-8, Latin-8…) on ne peut lire directement les caractères d'un fichier car on ne peut garantir que ces derniers utilisent le même codage que nous.

2. Compréhension des demandes initiales du cahier des charges

Ainsi le fait de considérer tous les fichiers comme binaires fait partie des demandes les plus importantes du cahier des charges. Ce dernier indique aussi la création de deux programmes distincts pour compresser et décompresser. De plus nous devrons faire attention aux éléments suivants :

- Organiser le sujet en module
- Utiliser de types de données logiques et adaptées
- Avoir un code optimisé et efficace

3. Répartitions du travail

Dès le début nous avons fait le choix de se répartir le travail entre nous. Ainsi Nathan s'occuperait du programme de compression et Aïcha de celui de décompression. Aussi nous nous sommes accordés sur un travail commun sur les différents modules. En effet, comme les modules vont être utilisés par les deux programmes, il fallait un travail exclusivement commun pour s'adapter aux exigences et contraintes de chacun avec efficacité. Nous nous sommes constamment informés sur les avancées de chacun afin de pouvoir s'aider mutuellement lors de la rencontre de difficultés.

II. Travail sur le projet

1. Définition des structures logiques utilisées

Nous avons décidé d'utiliser deux structures principales: une structure d'arbre dite Tree et une structure de liste chaînée dite LCA. Les deux structures seront utilisées à l'aide de pointeurs et seront des structures limitées privées permettant une grande flexibilité sur les types de variables contenues à l'intérieur.

- La première Tree semblait plus que nécessaire car aucune autre structure existante ne semblait pouvoir s'adapter pleinement à l'utilisation et à la création d'un arbre de Huffman. Chaque type tree peut être une feuille ou noeud d'un arbre de Huffman, aussi chaque Tree est une cellule composée d'un identifiant (id : T_id), d'une donnée (Data : T_Data) et d'un arbre de gauche et un arbre de droite (right, left : T_Tree). Un nœud aura ainsi des arbres sur left et right alors qu'une feuille n'aura rien. Dans le cas de son utilisation dans la compression, T_id contiendra un élément présent dans le texte dans le cas d'une feuille et rien dans le cas d'un nœud. Enfin, Data contiendra l'occurrence d'un élément dans le cas d'une feuille et la somme des occurrences de toutes les feuilles en dessous pour un nœud donné.

```
TYPE T_Tree  EST POINTEUR VERS T_Node

TYPE T_Node EST UN ENREGISTREMENT
        Left : P_Node
        Right : P_Node
        Data : T_data
        id : T_id
FIN ENREGISTREMENT
```

- La deuxième structure, LCA, semblait aussi essentielle pour plusieurs raisons. Notamment il était évident qu'il faudrait dès le début stocker les éléments d'un texte en entrée dans une structure pour ensuite les traiter. La LCA est une structure contenant une Clé et une donnée génériques, ainsi que le pointeur vers la LCA suivante. Nous instancions au fil du projet plusieurs package de LCA en fonction des nécessités rencontrées durant le codage.

```
TYPE T_LCA is access T_Cellule;

TYPE T_Cellule EST UN ENREGISTREMENT
        Cle: T_Cle;
        Donnee: T_Donnee;
        Suivant: T_LCA;
 FIN ENREGISTREMENT
```

2. Premiers Niveaux des raffinages

2.1. Compresser

R0 : Compresser un fichier txt

R1 : Comment “Compresser un fichier txt” ?

- Lire un fichier txt Liste_Octet : in
- Compter le nombre d’occurence de chaque caractère Liste_Octet : in out LCA_Integer_Octet
- Construire l’arbre de Huffman Liste_Tree : out LCA_Integer_Tree; Liste_Octet : in out
- Enregistrer le Codage de Huffman Code : in out Unbounded_String

R2 : Comment “Lire un fichier txt” ?

Pour i allant de 1 au nombre de caractères
ajouter à texte le ième caractère

R2 : Compter le nombre d'occurrence de chaque caractère ?

- Pour i allant de 1 à Taille(texte)
  - Enregistrer Node avec lettre en donnée et id incrémenté de 1
  - Ajouter/remplacer ce Node dans la lca

R2 : Comment construire l’arbre de Huffman ?

Tant que taille(lca) >1 faire
- Chercher les deux plus petites clés
- Créer un noeud dans la lca avec comme fils gauche et fils droit les deux plus petites clés (freq de fg <= freq de fd)
- Supprimer les noeuds ayant ces deux plus petites clés de la lca

Fin Tant que

R2: Comment “Enregistrer le Codage de Huffman”?

- Parcourir l’arbre par la gauche jusqu'à tomber sur une feuille
- Enregistrer le caractère et son codage
- Remonter au noeud à droite précédent et refaire l’opération

2.2. Décompresser

R0: Décompresser un fichier .hff

R1: Comment " Décompresser un fichier .hff"?

- Lire un fichier .hff
  - File_name: Chaîne de caractères
- Enregistrer les octets représentant les caractères
  - Lca_Char : out T_LCA
- Stocker la suite du fichier dans une variable
  - Suite : Chaîne de caractères
- Enregistrer le parcours infixe
  - Lca_parcours : out T_LCA
- Reconstruire l'arbre de Huffman
  - Lca_Char, Lca_parcours : in T_LCA
  - Created_tree: out T_Tree
- Enregistrer le codage de Huffman correspondant à chaque caractère
  - Lca_Char_Code : out T_LCA
  - Created_Tree: in T_tree
- Ecrire le texte décodé dans un fichier texte
  - Lca_Char_Code : in T_LCA

R2: Comment "Lire un fichier .hff"?

Ouvrir(File_Name)

R2: Comment "Enregistrer les octets représentant les caractères"?

- Récupération de la position du caractèrere de fin
  - Pos_Caractere_fin : out T_Octet
- Enregistrer les octets correspondants aux caractères dans une liste chaîné
  - Octet_Courant : in out T_Octet
  - Octet_Precedent : in out T_Octet
  - Nb_Characters : in out Integer
  - Lca_Char : out T_LCA

R2: Comment "Stocker la suite du fichier dans une variable"?

Suite<- "";
Tant Que not EOF(File) Faire

- Lire un octet
  - Octet_Courant : out T_Octet
- Récupérer les bits de l'octet
  - Octet_Courant : in out T_Octet
  - Bit : in out T_octet
- Ajouter à suite "1" ou "0" suivant les valeur de bits
  - Suite : out chaîne de caractère

Fin Tant Que

R2: Comment "Enregistrer le parcours infixe"?

index <- 1
cpt <- 1 - - compteur des 1 rencontrés
LCA_Integer_data.initialiser(lca_id);

```
Tant que cpt <= nb_char Faire
        Si (Suite)(index) ='1' Alors
                LCA_Integer_data.enregistrer(lca_parcours,index, 1);
                cpt <- cpt +1
        Sinon
                LCA_Integer_data.enregistrer(lca_parcours,index,0);
        FinSi
         index <- index+1;
Fin Tant que
```

- Supprimer les caractères lus de la chaîne de caractères
  Suite : in out Chaîne de caractères
  Index : in Integer

R2: Comment “Reconstruire l'arbre de Huffman”?
Utilisation de la procédure Huffman Tree

R2: Comment “Enregistrer le codage de Huffman correspondant à chaque caractère”?
Utilisation de la procédure Huffman Code

R2: Comment écrire le texte décodé dans un fichier texte ”?
code : chaîne de caractères

```
indice_lecture<- 1;
Pour i allant de 1 à Length(Suite) Faire
        code <- Suite(indice_lecture..i);
```

- Chercher si code figure comme clé

```
Fin Pour
```

3. Modules fondamentaux

3.1. Module Tree, fonctions et procédures essentielles

– Créer un noeud dans un Tree

```
procedure Create_Node (Tree : out T_tree; left :  in T_Tree; right : in T_Tree; id : in T_id;
data: in T_data ) is
begin
        Tree := new T_Node'(left,right,data, id);
end Create_Node;
```

–Créer une feuille dans un tree

```
procedure Create_Leaf (Tree : out T_Tree ; Id : in T_id; Data : in T_data) is
begin
        Tree := new T_Node'(null,null,Data,Id);
end Create_Leaf;
```

– Donner en sortie le parcours infixe de Tree

```
procedure Parcours_Infixe( Tree : in T_Tree ; infixe : out Unbounded_String) is
begin
        if not is_empty(left (tree)) then
                infixe := infixe & To_Unbounded_String("0") ;
                Parcours_Infixe(tree.all.left, Infixe);
        end if;
        if not is_empty(right (tree)) then
                infixe :=  infixe & To_Unbounded_String("1");
                Parcours_Infixe(tree.all.right, Infixe);
        end if;
end Parcours_Infixe;
```

Remarquons qu'il n'y a aucune fonction permettant de donner le codage de Huffman dans le module tree. En effet au vu des différences de fonctionnement entre le programme compresser et décompresser et de la variété des types de variables utilisées dans les différents code il a été décidé d'ajouter une fonction Huffman_Code dans les programmes directement au lieu d'en créer une générique dans le module Tree. Il en était de même pour la procédure permettant d'afficher l'arbre et celle permettant de la reconstruire.

3.2. Module LCA, fonctions et procédures essentielles

Nous avons directement utilisé le module LCA codé précédemment en TP.

On parcourt récursivement la liste jusqu'à trouver l'élément dont la clé est égale à la clé recherchée. Si on la trouve, on change sa donnée.. Sinon on crée un nouvel élément

```
procedure Enregistrer (Sda : in out T_LCA ; Cle : in T_Cle ; Donnee : in T_Donnee)
begin
        if Est_Vide(Sda) then
                Sda := new T_Cellule'(Cle, Donnee, NULL);
        elsif Sda.All.Cle = Cle then
                Sda.All.Donnee :=Donnee;
        else
                Enregistrer (Sda.All.Suivant, Cle, Donnee);
        end if;
end Enregistrer;
```

On parcourt récursivement la liste jusqu'à trouver l'élément dont la clé est égale à la clé recherchée. Si on la trouve, on change sa donnée.. Sinon on crée un nouvel élément

```
function La_Donnee (Sda : in T_LCA ; Cle : in T_Cle) return T_Donnee is
begin
        if Est_Vide(Sda) then
                raise Cle_Absente_Exception;
                elsif Sda.All.Cle=Cle then
                return Sda.All.Donnee;
        else
                return La_Donnee(Sda.All.Suivant, Cle);
        end if;
end La_Donnee;
```

Ajouter au début d'une LCA (contrairement à enregistrer)

```
procedure Ajouter_au_debut (Sda : in out T_LCA ; Cle : in T_Cle ; Donnee : in
T_Donnee) is
begin
        Sda := new T_Cellule'(Cle, Donnee, Sda);
end Ajouter_au_debut;
```

Doubler le dernier élément d'une lca

```
procedure Double (Sda : in out T_LCA) is
begin
        if Sda.all.suivant = null then
                Sda.all.suivant := New T_Cellule'(Sda.all.Cle, Sda.all.Donnee, NULL);
        else
                double(Sda.all.suivant);
        end if;
end double;
```

4. Travail sur les différents programmes

4.1. Compresser

Comme dit précédemment, on considère tous les fichiers textes comme des fichiers binaires. Ainsi au lieu de créer un arbre avec des caractères en id le choix à été fait pour le module compresser de créer un arbre avec les octets lus dans le fichier source. Ainsi, comme chaque caractère est codé sur 8 bit, chaque octet de l'arbre correspond a un caractère du texte. Comme on ne connaît effectivement pas le codage utilisé par le fichier source (UTF-8, LATIN-8…) peut importe quel caractère est associé à tel ou tel octet, on sait juste qu' un octet est un des caractères du texte, ce qui est suffisant pour créer un arbre de Huffman.

Il a été décidé dans son fonctionnement global de d'abord lire le fichier source et d'enregistrer dans un LCA type LCA_Integer_Octet le nombre de fois où chaque octet est présent dans le texte. Ensuite on convertit toutes ces données en une LCA type LCA_Integer_Tree ou chaque donnée est une feuille de l'arbre avec son octet et son nombre d'occurrence.

Enfin on parcourt cette LCA grâce à une fonction Huffman_Tree qui va créer un nœud avec les deux feuilles/nœuds ayant le moins d'occurrence puis les supprime. Ainsi après exécution de la procédure nous obtenons l'arbre de Huffman.

```
procedure Huffman_Tree (Liste_Tree : in out LCA_Integer_Tree.T_Lca ; Tree : out T_Tree)
is
        Tree_g : T_Tree;
        Tree_d : T_Tree;
        Node : T_Tree;
        Cle : Integer;
        Taille : Integer := LCA_Integer_Tree.Taille(Liste_Tree);
begin
        while LCA_Integer_Tree.Taille(Liste_Tree)>1 loop
                Create_Node([..] trop long pour être inclus ici); --on suppose que la 1er valeur
est la plus petite
                Cle := La_Cle(Liste_Tree);
                min_Tree(Liste_Tree,Tree_g,Cle); --Tree_g est le vrai minimum
                Supprimer(Liste_Tree,Cle);

                Create_Node([..] trop long pour être inclus ici); --on suppose que la 1er valeur
est la plus petite
                Cle := La_Cle(Liste_Tree);
                min_Tree(Liste_Tree,Tree_d,Cle); --Tree_d est le vrai deuxième minimum
                        Supprimer(Liste_Tree,Cle);


                Create_Node(Node,Tree_g,Tree_d,T_Octet(27),data(Tree_g)+data(Tree_d));
                        Taille := Taille+1;
```

```
            Enregistrer(Liste_Tree,Taille,Node);
            end loop;
        Create_Node(Tree,left(Node),right(Node),T_Octet(27),data(Node));
end Huffman_Tree;
```

La fonction min_Tree trouve le noeud/feuille avec le moins d'occurrence dans la liste et renvoi le noeud/feuille correspondant ainsi que sa clé dans la LCA.

Il aurait été effectivement plus rapide de créer directement une LCA de type LCA_Integer_Tree et de travailler dessus directement mais par soucis de simplicité et par manque de temps, la solution la plus simple à mettre en oeuvre fut d'abord de créer une liste avec les octets et leurs occurrences puis de les convertir en feuille après coup comme expliqué ci dessus.

4.2. Décompresser

La décompression reposait nécessairement sur la compression des fichiers et la structure du fichier compressé . Nous avons alors décidé que le fichier compressé contiendrait dans l'ordre: la position du caractère de fin dans l'arbre de huffman, les octets correspondants aux caractères lus dans le fichier dans l'ordre de leur apparition dans l'arbre de Huffman en doublant le dernier, le parcours infixe de l'arbre d'Huffman en ajoutant un 1 à la fin et finalement le texte traduit en utilisant la table de Huffman.

Suivant cette structure, lors de la décompression, nous commençons par récupérer la position du caractère de fin et stocker sa valeur dans la variable Position_caractere_fin. Nous procédons ensuite à la lecture des octets correspondants aux caractères présents dans le fichier qu'on stocke dans une liste simplement chaînée dont les clés sont des entiers et les données des octets. Une fois deux octets successifs égaux, nous savons que la liste des caractères est finie et nous passons au traitement du reste du contenu.
A ce point-là, la partie non parcourue du fichier compressé passé en ligne de commande contient le parcours infixe et le texte traduit selon la table de Huffman. Les deux ne constituant pas des données réparties en octets, nous avons opté pour une autre alternative pour les analyser. Nous procédons d'abord au stockage des octets dans une chaîne de caractères (suite de un et de zéro), ce qui nous amène à l'application de l'algorithme cité dans le sujet du projet permettant de récupérer les bits d'un octet.

Après cela, nous utilisons une autre liste simplement chaînée dont les clés sont également des entiers et les données sont des chaînes de caractères pour stocker le parcours infixe de l'arbre d'Huffman. En comparant le nombre de uns lus aux nombres de caractères nous pouvons détecter la fin du parcours infixe (Le nombre de un dans un parcours infixe est égale au nombre de caractères).
C'est ici que nous procédons à la reconstruction de l'arbre de Huffman à partir des deux listes: l'une contenant les octets (caractères) et l'autre le parcours infixe à l'aide de la procédure Huffman Tree définie dans le module tree.
Une fois l'arbre créée, nous obtenons le codage Huffman des caractères en faisant appel à la procédure Huffman Code que nous allons alors stocker dans une liste simplement chaînée dont les clés seront des chaînes de caractères correspondants au codage Huffman et les données seront également des chaînes de caractères avec les octets correspondants au codage.

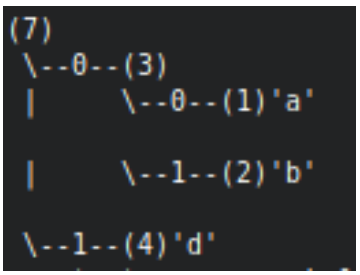
Il ne nous reste plus qu'à traduire le texte en parcourant les caractères restants dans la chaîne de caractères suite de un et de zéro. Pour cela, nous utilisons une variable intermédiaire à laquelle nous affectons le premier caractère de la suite. Nous le cherchons s'il figure comme clé dans notre liste. Si oui, alors nous écrivons la donnée correspondante à cette clé dans le fichier décompressé et nous affectons la chaîne de caractère vide à la variable intermédiaire. Sinon nous y ajoutons le prochain caractère. Nous répétons le même processus jusqu'à avoir lu la totalité de la suite de 1 et de 0.
Ainsi nous obtenons le fichier décompressé!

5. Premiers tests

Au vu de l'avancée des modules et des programmes compresser et décompresser, les premiers tests peuvent avoir lieu. Les test des LCA ont été récupérés du fichier du TP réalisé précédemment : on teste l'ajout, la suppression, etc.
Il était surtout intéressant de tester les différentes procédures et fonctions implémentées dans le module tree. Ainsi nous avons ajouté au fur et à mesure des tests permettant de vérifier les sous-programmes. Comme par exemple pour la création d'arbres, nous avons cherché à vérifier si notre structure était vraiment adaptée. Ainsi le fichier test_tree.adb crée manuellement un arbre pour un texte contenant un a, deux b et quatre d. Nous créons alors manuellement les feuilles :

Noeud1 ← feuille de a(coté left) et feuille de b(coté right) : occurence = 3
Noeud2 ← Noeuds1(coté left) et feuille de d (coté right): occurence =7

On en profite aussi pour tester la fonction Print_Tree et on obtient le résultats suivant:



On obtient le même résultat en faisant l'arbre sur feuille. Ainsi grâce à ce cas particulier testé ici on peut déjà affirmer que la structure de l'arbre fonctionne correctement.

Nous réalisons aussi le test du parcours infixe et du codage Huffman de chaque caractères et nous comparons les résultats avec ceux trouvés sur feuille.

III. Développement du travail et résolutions des problèmes

1. Problèmes majeurs rencontrés et solutions trouvées

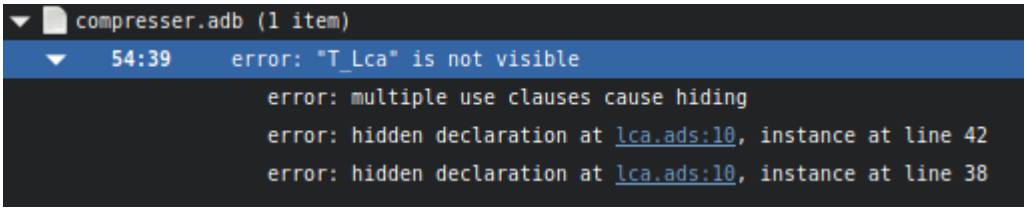
1.1. Compresser

Un problème majeur était la gestion d'un programme contenant plusieurs package pour un même type générique. En effet un tel programme n'a jamais été codé en TP, en essayant de définir deux packages :

```
package LCA_Integer_Octet is
        new LCA (T_Cle => T_Octet, T_Donnee => Integer);
use LCA_Integer_Octet;

package LCA_Integer_Tree is
        new LCA (T_Cle => Integer, T_Donnee => T_Tree);
use LCA_Integer_Tree;
```

Le compilateur renvoyait l'erreur :



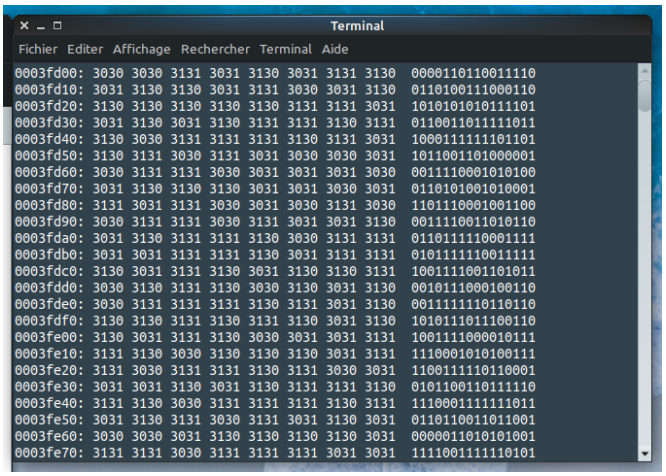
Cela semblait en effet logique, comment le compilateur pouvait comprendre de quel T_LCA il s'agit sachant qu'il y en deux ? La première idée fut d'abandonner les deux packages de LCA, n'en gardant qu'une, ce problème ne se poserait plus. Néanmoins cela compliquerait grandement le codage existant obligeant à revoir en profondeur le fonctionnement global de compresser.adb. Après concertation entre binôme nous avons trouvé que la solution était de simplement préciser à chaque appel de LCA de quel package on parlait. Ainsi au lieu d'écrire :

Liste : in out T_LCA on écrit Liste : in out LCA_Integer_Octet.T_Lca

Une autre difficulté majeure dans compresser fut de réécrire les codages de Huffman de chaque caractère dans le fichier de sortie. En effet on crée une variable code : Unbounded_String contenant le code à écrire en binaire avec :

```
T_String'write(S_compressed,To_String(Code))
```

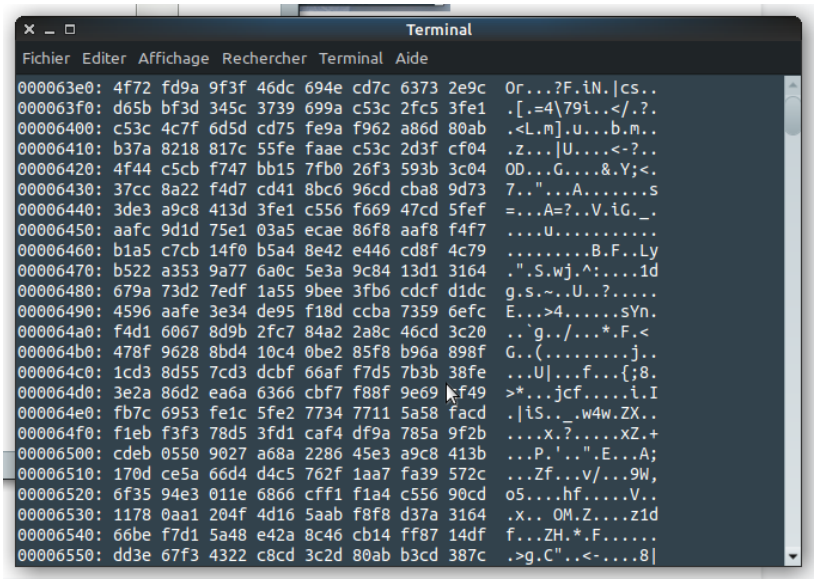
Au début nous pensions que le code marchait car on obtient bien un fichier binaire en sortie. En revanche on s'est vite rendu compte que le fichier de sortie était plus lourd que le fichier d'origine. En regardant le code du fichier sortant avec xxd on obtenait cela :



On se rend compte que le programme écrit les caractères 1 et 0 (30 est le code Hexadécimal de 0 et 31 celui de 1). En fait le programme n'écrit pas des octets mais des caractères, donc chaque 0 ou 1 est codé sur 8 bits prenant une place conséquente.
On décide alors de reprendre la partie du code écrivant dans le fichier et utilisons plutôt :

```
for i in 1..(length(Code)/8) loop
    IntVal := Integer'Value("2#"&To_String(Code)(i*8-7..8*i)&"#");
    T_Octet'write(S_compressed,T_Octet(IntVal));
end loop;
```

Ici la boucle permet d'écrire les bits que 8 par 8 (donc on forme un octet). Nous avons évidemment préalablement ajouter des 0 a la fin de Code pour avoir length(Code) comme un multiple de 8.
IntVal permet d'obtenir la valeur entière de chaque suite de 8 bits. Enfin T_Octet avec write écrit bien le codage en octet de l'entier en entrée.
En utilisant xxd on obtient cette fois ci :



On code ici correctement les octets.

2.2. Décompresser

Au fil du développement du projet, j'ai dû implanter plusieus procédures et/ou fonctions permettant de manipuler les arbres de Huffman dont une utilisée dans la décompression : Huffaman_tree. J'ai été amené à concevoir une procédure permettant de reconstruire un arbre de Huffman à partir d'un parcours infixe et d'une liste de données, chose qui n'était pas très facile. Après plusieurs versions, j'ai pu la faire.

L'exigence de lecture en octet (8 bits) n'était pas pratique pour le traitement du parcours infixe et du texte codé. C'est pourquoi, j'ai choisi de retranscrire les bits dans une chaîne de caractères plus facile à manipuler.

IV. Finalisation du projet

1. Peaufinage et mise en place de l'interface utilisateurs

Les codes fonctionnent à présent. Le travail est maintenant de mettre en forme le code (indentation, commentaires) et de gérer l'interface utilisateur. Les deux programmes seront lancés depuis un terminal. Le premier argument pourra être -b pour le mode bavard. Le deuxième argument ou le premier (cas sans -b) sera le fichier source.
On gère ainsi plusieurs exceptions pour aider l'utilisateur en cas de problème par exmple dans compresser nous avons :

```
when ADA.IO_EXCEPTIONS.NAME_ERROR =>
        New_Line;
        Put_line("!!! Erreur fichier inexistant !!!") ;
        New_Line;

when COMPRESSER.MISSING_ARGUMENT =>
        New_Line;
        Put_line("!!! Erreur, absence d'argument. Exemple syntaxe : ./compresser -b
fichier.txt !!!") ;
        New_Line;

when COMPRESSER.TOO_MANY_ARGUMENT =>
        New_Line;
        Put_line("!!! Erreur trop d'arguments en entree !!!") ;
        New_Line;

when COMPRESSER.UNKNOWN_ARGUMENT =>
        New_Line;
        Put_line("!!! Erreur argument inconnu !!!") ;
        New_Line;
```

2. Tests finaux

2.1. Compresser

On teste maintenant le programme en se plaçant à la place de l'utilisateur. On voit que les exceptions fonctionnent pour tous les cas d'erreur courants. On remarque surtout que pour un fichier donné, le fichier compressé prend maintenant presque 2 fois moins de place.
On considère ici que Compresser.adb fonctionne correctement et est utilisable par un utilisateur.

2.2. Décompresser

Nous pouvons tester la décompression en compressant d'abord un fichier texte par exemple. Après décompression, nous obtenons exactement le même fichier initialement compressé.

3. Améliorations et évolutions

3.1. Compresser

Première remarque, le programme est lent. En effet pour de gros fichiers textes la compression peut mettre jusqu'à quelques secondes. Ceci s'explique tout simplement à cause d'une mauvaise optimisation du code. En effet le code contient beaucoup de boucles, voire de boucles imbriquées dans d'autres, ce qui ralentit considérablement son exécution avec de lourds fichiers. L'optimisation du code est donc la prochaine priorité dans la perspective d'une future évolution du projet, par manque de temps elle n'a pu être que relative.

Aussi le programme ne peut prendre qu'un seul fichier en entrée à la fois, rendant impossible la compression de multiples fichiers automatiquement. Il faudrait ainsi prendre en compte plus d'arguments lors de l'exécution.

3.2. Décompresser

Les deux procédures Huffman Code et print tree sont utilisées à plusieurs reprises, cependant, elles ne figurent pas dans le module tree parce qu'elles ne sont pas génériques. Nous faisons alors de la duplication de code qui pourrait être éviter. Nous pouvons donc améliorer le code en la modifiant pour qu'elle soit générique.
Nous pouvons également ajouter des exceptions pour signaler tout éventuel cas d'erreur.
Par ailleurs, nous pourrons faire évoluer le projet pour qu'il puisse manipuler tout type de fichier (txt, jpg, …).

4. Bilan personnel de fin de projet

4.3. Nathan

J'étais personnellement très intéressé par ce sujet. Ayant travaillé sur la compression d'image avec perte pour mon TIPE j'avais aussi vu l'existence de méthodes sans pertes utilisant notamment l'arbre de Huffman mais n'avait pas travaillé dessus bien qu'intéressé. Ce projet m'a permis de comprendre son fonctionnement fondamental.
J'ai passé quelques après midi des vacances sur le projet, au final seulement quelques heures ont été nécessaires pour comprendre le sujet et coder les structures qui allaient êtres utilisées. En revanche coder le programme principal à prit de nombreux jours, notamment pour résoudre des problèmes de syntaxe ou de logique. Le rapport aura demandé quelques jours pour être rédigé.
Ce projet m'a personnellement permis de comprendre les bases de la compression de données qui est une opération courante en informatique. Aussi j'ai pu comprendre la structure des données enregistrées dans un fichier et ai pu retravailler la programmation orientée objet avec des pointeurs.

4.4. Aïcha

Ce projet m'a permis de mettre en pratique les connaissances que j'ai acquises dans une UE d'algorithmique des structures arborescentes en licence informatique. Ayant traité plusieurs types d'arbres (ABR, AVL, Arbre Rouge et Noir …), j'avais en effet déjà des idées de manipulation de la structure arborescentes. J'ai pu également me familiariser davantage avec la notion de module et surtout l'instanciation en Ada.
La compréhension du sujet et des idées d'algorithmes pour le résoudre n'était pas difficile. Avec mon binôme, nous avions déjà une idée claire de l'algorithme à adopter pour la compression et la décompression