Programmation Impérative

Rapport du projet : Codage de Huffman

Réalisé par :

Aya RIFAI
Fadwa ELLAIK
Groupe CD14

Résumé

L'objectif de ce rapport est de résumer les différentes étapes de ce projet. Nous allons présenter les types auxquels nous avons recouru et leur intérêt pour l'implémentation. Nous allons aussi expliquer les algorithmes utilisés pour les programmes de compression et de décompression . Nous allons aussi présenter l'avancement de notre travail et les difficultés rencontrées lors de la réalisation .

Table des matières

Introduction

La compression des données est un enjeu fondamental dans le domaine de l'informatique, permettant de réduire la taille des fichiers tout en préservant leur contenu. Parmi les techniques de compression les plus utilisées, le codage de Huffman occupe une place centrale. Ce procédé repose sur un algorithme qui associe à chaque symbole d'un fichier une représentation binaire de longueur variable, proportionnelle à la fréquence d'apparition de ce symbole. Grâce à cette approche, il est possible de minimiser la taille totale des données tout en maintenant une compression sans perte, où les informations originales peuvent être restituées intégralement. Dans le processus inverse, appelé décompression, le fichier compressé est décodé pour retrouver le contenu initial. Le codage de Huffman illustre ainsi l'équilibre entre efficacité algorithmique et optimisation des ressources, en jouant un rôle clé dans des domaines allant des systèmes de fichiers aux communications numériques.

1 Début du projet

1.1 Compréhension globale du sujet

L'objectif du sujet est de créer deux programmes principaux de compression et de décompression en utilisant le principe du codage de Huffman. L'idée pour la compression est d'utiliser l'arbre de Huffman construit à partir de la table de fréquences des différents caractères du texte , de construire la table de Huffman associant à chaque caractère son code correspondant à partir de l'arbre de Huffman , et réécrire le texte en utilisant les codes obtenus. La décompression est l'opération inverse qui consiste à reconstruire l'arbre de Huffman à partir des informations présentes dans le fichier compressé , et restituer les données du fichier original.

1.2 Compréhension des exigences du cahier de charges

Pour réaliser le travail demandé, il était nécessaire d'utiliser les modules déjà vus en TP et en TD qui garantissent une bonne approche du problème . Il fallait aussi opter pour des codes efficaces et simples à comprendre.

2 Architecture de l'application en modules

2.1 Description des modules

2.1.1 Module ABR

Le module ABR est un module qui nous a permis l'implémentation d'un arbre dans Ada. Il était fondamental pour les deux programmes , puisque la construction de l'arbre de Huffman reposait sur la bonne implémentation de ce module . C'est une structure de données associative qui utilise un pointeur sur un enregistrement.Elle est définie de la manière suivante :

```
TYPE T_Noeud
TYPE T_ABR EST POINTEUR SUR T_Noeud
TYPE T_Noeud EST ENREGISTREMENT
    Symbole : T_Symbole
    Frequence : T_Frequence
    Gauche : T_ABR
    Droit : T_ABR
    FIN ENREGISTREMENT
```

Ici, T_Symbole et T_Frequence sont des types génériques que nous avont définis afin de généraliser l'utilisation du module ABR. Nous allons les spécifier par la suite dans nos programmes compresser et décompresser . L'arbre ainsi défini contient pour chaque noeud un symbole, une fréquence et deux fils droit et gauche qui peuvent éventuellement être nuls.

2.1.2 Module LCA

Nous avons également utilisé le module LCA des listes chaînées associatives qui s'est avéré important dans l'implémentation de la table de fréquences et qui facilitait le parcours et l'enregistrement des différents caractères présents dans le fichier texte à compresser.Une liste chaînée associative est un pointeur sur un enregistrement . Elle est définie comme suit :

```
TYPE T_Cellule
TYPE T_LCA EST POINTEUR SUR T_Cellule
TYPE T_Cellule EST ENREGISTREMENT
    Cle : T_Cle
    Valeur : T_Valeur
    Suivant : T_LCA
    FIN ENREGISTREMENT
```

Comme précédemment, les types T_Cle et T_Valeur sont définis de manière générique pour généraliser leur utilisation .

2.2 Principales fonctions et procédures des modules

2.2.1 Module ABR

Fonction Est_Feuille

Cette fonction détermine si un noeud est une feuille.Elle retourne ainsi un booléen :

```
function Est_Feuille(Noeud : in T_ABR) return Boolean is
begin
    return (Noeud.All.Gauche = Null) and (Noeud.All.Droit =
       Null);
end;
```

Procedure Creer_Nouveau_Noeud

Cette procédure a été définie suite à la nature privée du type T_ABR tel que nous l'avons défini. Elle sert alors à créer un nouveau noeud à partir d'un symbole et d'une fréquence donnés de la manière suivante :

```
procedure Creer_Nouveau_Noeud(Symbole : in T_Symbole ;
   Frequence : in T_Frequence ; Abr_Gauche : in T_ABR ;
   Abr_Droit : in T_ABR;Nouveau_Noeud : out T_ABR ) is
begin
    Nouveau_Noeud := new T_Noeud'(Symbole,Frequence,
       Abr_Gauche,Abr_Droit);
end Creer_Nouveau_Noeud;
```

De la même manière que Creer_Nouveau_Noeud , nous avons défini des fonctions comme Symbole,Valeur,Gauche et Droit qui retournent respectivement le symbole, la valeur , le fils gauche et le fils droit d'un arbre donné en entrée. Ces fonctions ont été définis suites aux erreurs générées en essayant de manipuler le type T_ABR privé à l'extérieur du corps du module.

2.2.2 Module LCA

Procedure Enregistrer

Cette procédure permet d'enregistrer une valeur dans la liste chaînée associative . Elle l'enregistre en l'associant à la clé Cle si elle existe , sinon , elle crée une nouvelle cellule contenant la clé et la valeur associée.

```
procedure Enregistrer(Sda : in out T_LCA; Cle : in T_Cle ;
   Valeur : in T_Valeur) is
    Pointeur : T_LCA;
    d : T_LCA;
    function dernier(Sda : in T_LCA) return T_LCA is
        p : T_LCA;
    begin
        p := Sda;
        while p.All.Suivant /= Null loop
            p := p.All.Suivant;
        end loop;
        return p;
    end dernier;
begin
   if not Cle_Presente(Sda,Cle) then
        Pointeur := new T_Cellule'(Cle,Valeur,Null);
        if Sda = Null then
            Sda := Pointeur;
        else
            d := dernier(Sda);
            d.All.Suivant := Pointeur;
        end if;
    else
        Pointeur := Sda;
        while Pointeur /= Null loop
            if Pointeur.All.Cle = Cle then
                Pointeur.All.Valeur := Valeur;
                exit;
            else
                Pointeur := Pointeur.All.Suivant;
            end if;
        end loop;
    end if;
end Enregistrer;
```

Fonction Cle_Presente

Cette fonction est nécessaire pour faire le parcours de la LCA et déterminer à l'aide du booléen qu'elle retourne si la clé est présente .

```
function Cle_Presente(Sda : in T_LCA ; Cle : in T_Cle) return
    Boolean is
    Liste : T_LCA;
begin
    Liste := Sda;
    while not (Liste = Null) loop
        if Liste.All.Cle = Cle then
            return True;
        end if;
        Liste := Liste.All.Suivant;
    end loop;
    return False;
end Cle_Presente;
```

Fonction La_Valeur

Cette fonction , étant donné une Sda et une Clé, retourne la valeur associée à cette clé à condition que la clé soit présente dans la Sda , sinon , elle génère une erreur.

```
function La_Valeur(Sda : in T_LCA;Cle : in T_Cle) return
   T_Valeur is
begin
    if Sda = Null then
        raise Cle_Absente_Exception;
    elsif Sda.All.Ce = Cle then
        return Sda.All.Valeur ;
    else
        return La_Valeur(Sda.All.Suivant,Cle);
    end if;
end La_Valeur;
```

2.3 Présentation des principaux algorithmes

2.3.1 Compression

Dans l'algorithme de compression, nous avons recouru à plusieurs fonctions et procédures intermediaires. Le but était de construire d'abord la table de fréquences des caractères du texte , tout en ajoutant le symbole "$" qui marque la fin du texte.
Ensuite , nous avons procédé à la construction de l'arbre de Huffman , pour cela , nous avons utilisé des fonctions et procédures telles que la fonction **fusion** qui permet de fusionner deux noeuds en un arbre , la procédure **Trier_Noeuds** qui trie une liste de noeuds (nous avons défini la liste des noeuds comme un tableau de capacité 257 vu que le nombre maximal de caractères est 256 et nous avons aussi ajouté le caractère spécial "$" marquant la fin du texte) du noeud de fréquence minimale au noeud de fréquence maximale . Cette procédure nous facilite l'opération de fusion puisqu'à chaque étape , les deux noeuds situés au début de la liste des noeuds sont ceux à fusionner.Nous avons aussi utilisé la procédure **Construire_Feuilles** qui, à partir de la table de fréquences obtenur, construit la liste des noeuds mentionnée dessus.
L'implémentation de ces fonctions sera réduite dans ce qui suit et nous ne laisserons que l'implémentation de l'arbre de Huffman pour ne pas encombrer.

```
procedure Construire_Arbre_Huffman(Arbre_Huffman : out
   ABR_UStr_Int.T_ABR; Table_Frequences : in LCA_UStr_Int.
   T_LCA) is
    type T_Liste_Noeuds is array(1..257) of ABR_UStr_Int.
       T_ABR;
    Liste_Noeuds : T_Liste_Noeuds;
    Nombre_Noeuds : Integer := 0;
    Arbre_Temporaire : ABR_UStr_Int.T_ABR;
    Arbre_Null : ABR_UStr_Int.T_ABR;
begin
    Initialiser(Arbre_Null);
    Construire_Feuilles(Table_Frequences, Liste_Noeuds,
       Nombre_Noeuds);
    Nombre_Noeuds := Nombre_Noeuds + 1;
    Creer_Nouveau_Noeud(Liste_Noeuds(Nombre_Noeuds),
       To_Unbounded_String("\$"),0,Arbre_Null,Arbre_Null);
    while Nombre_Noeuds > 1 loop
        Trier_Noeuds(Liste_Noeuds,Nombre_Noeuds);
        Remplacer(Fusion(Liste_Noeuds(1),Liste_Noeuds(2)),
           Arbre_Temporaire);
        for I in 3..Nombre_Noeuds loop
            Liste_Noeuds(I-2) := Liste_Noeuds(I);
        end loop;
        Nombre_Noeuds := Nombre_Noeuds-1;
        Remplacer(Arbre_Temporaire,Liste_Noeuds(Nombre_Noeuds
           ));
    end loop;
    Remplacer(Liste_Noeuds(1),Arbre_Huffman);
```

```
end Construire_Arbre_Huffman;
```

Après avoir créé l'arbre de Huffman, nous avons procédé à la création de la table de Huffman . Ainsi, nous avons implémenté la procédure **Table_De_Huffman** qui la construit. Nous nous sommes aidées dans la construction de la procédure **Generer_Codes** qui effectue un parcours en profondeur de l'arbre de Huffman et associe ainsi à chaque symbole son code correspondant de la manière suivante:

```
procedure Table_De_Huffman(Arbre_Huffman : in ABR_UStr_Int.
   T_ABR ; Table_Huffman : out LCA_UStr_Ustr.T_LCA) is
    procedure Gnerer_Codes(Arbre : in ABR_UStr_Int.T_ABR;
       Code : in Unbounded_String; Table : in out
       LCA_UStr_UStr.T_LCA) is
        Code_G, Code_D : Unbounded_String;
    begin
        if Est_Feuille(Arbre) then
            Enregistrer(Table,Symbole(Arbre),Code);
        else
            Code_G := Code & To_Unbounded_String("0");
            Generer_Codes(Gauche(Arbre),Code_G,Table);
            Code_D := Code & To_Unbounded_String("1");
            Generer_Codes(Droit(Arbre),Code_D,Table);
        end if;
    end Generer_Codes;
    Table : LCA_UStr_UStr.T_LCA;
begin
    Initialiser(Table);
    Generer_Codes(Arbre_Huffman,To_Unbounded_String(""),Table
       );
    Remplacer(Table,Table_Huffman);
end Table_De_Huffman;
```

Ensuite, nous avons procédé à l'inclusion des différentes étapes dans le fichier de sortie compressé avec une extension ".hff" . Le contenu de ce fichier binaire est donc les différentes code ASCII des caractères présents dans le fichier d'origine . Nous avons respecté la méthode exigée qui consiste à remplacer le symbole "$" par sa position dans la table de Huffman , et à répéter le dernier symbole deux fois pour marquer la fin des symboles. Nous avons aussi inclu , comme demandé, dans le fichier compressé, la structure de l'arbre de Huffman ainsi que le texte encodé . Nous avons créé le texte encodé à partir du texte original à l'aide de la procédure **Encoder_Texte** . Bien sur, pour faire cela , nous avons instancié les différents modules mentionnés ci-dessus , et avons importé la bibliothèque Ada.Strings.Unbounded.

```
package ABR_UStr_Int is new ABR(T_Symbole => Unbounded_String
   , T_Frequence => Integer);
package LCA_UStr_Int is new LCA(T_Cle => Unbounded_String,
   T_Valeur => Integer);
package LCA_UStr_UStr is new LCA(T_Cle => Unbounded_String,
   T_Valeur => Unbounded_String);
```

Pour écrire le texte encodé en binaire, nous avons utilisé le type T_Octet défini comme suit :

```
type T_Octet is mod 2**8;
for T_Octet'Size use 8;
```

2.3.2 Décompression

Le programme de décompression basé sur le codage de Huffman a été conçu pour lire un fichier compressé, reconstruire l'arbre et la table de Huffman, et décoder les données compressées afin de restituer le fichier original. Cette démarche a été soigneusement découpée en plusieurs étapes sous forme de sous-programmes, chacun ayant un rôle précis dans la chaîne globale de décompression. Cette approche modulaire a permis d'aborder la complexité du projet de manière organisée, tout en facilitant le débogage et l'optimisation. Chaque sous-programme a été conçu et testé pour fonctionner de manière indépendante, tout en s'intégrant harmonieusement dans l'ensemble, garantissant ainsi la robustesse et l'efficacité du programme. D'abord, la procédure "Lire_Contenu_Fichier" remplit efficacement sa fonction en extrayant la liste des caractères uniques d'un fichier compressé et en générant un flux binaire pour la décompression. La gestion des octets, la conversion en caractères et en binaire, ainsi que la détection des doublons pour marquer la fin de la liste grâce à un tableau, sont des choix bien adaptés aux besoins du projet. Toutefois, il serait bénéfique d'optimiser le stockage des caractères en utilisant une structure de données associative plus performante (T_LCA), afin de réduire la complexité de la recherche.

```
procedure Lire_Contenu_Fichier(Nom_Fichier : in String;
   Tableau_Caracteres : out T_Liste_Caracteres;
   Nombre_Caracteres : in out Integer; Contenu : in out
   Unbounded_String) is
       Fichier  : Ada.Streams.Stream_IO.File_Type;
       Stream   : Stream_Access;
       Tableau : T_Liste_Caracteres := (others =>
          To_Unbounded_String(""));
       Octet : T_Octet;
       Indice_Dollar : Integer;
       Caractere : Character;
       Double : Boolean;
       Indice : Integer := 0;
       Bit : Integer;
    begin
       -- Ouvrir le fichier en lecture
        Open(File => Fichier, Mode => Ada.Streams.Stream_IO.
           In_File, Name => Nom_Fichier);
        Stream := Ada.Streams.Stream_IO.Stream(Fichier);
        T_Octet'Read(Stream, Octet);
        Indice_Dollar := Natural(Octet);
        Tableau(Indice_Dollar) := To_Unbounded_String("\$");
        loop
```

```
            T_Octet'Read(Stream, Octet);
            Caractere := Character'Val(Integer(Octet));
            Double := False;
            for J in 0 .. Indice loop
               if Tableau(J) = To_Unbounded_String(Character'
                  Image(Caractere)) then
                  Double := True;
               end if;
            end loop;

            if Double = False then
               if Tableau(Indice) = To_Unbounded_String("\$")
                   then
                  Indice := Indice + 1;
               end if;
               Tableau(Indice) := To_Unbounded_String(
                  Character'Image(Caractere));
               Nombre_Caracteres := Nombre_Caracteres + 1;
               Indice := Indice + 1;
            end if;
         exit when Double;
         end loop;
         Tableau_Caracteres := Tableau;


         while not Ada.Streams.Stream_IO.End_Of_File(Fichier)
            loop
            T_Octet'Read(Stream, Octet);

            for I in 1 .. 8 loop
               Bit := Integer(Octet) / 128;
               Octet := (Octet * 2) mod 255;
               if Bit = 1 then
                  Contenu := Contenu & To_Unbounded_String
                     ("1");
               else
                  Contenu := Contenu & To_Unbounded_String
                     ("0");
               end if;
            end loop;
         end loop;


      Close(Fichier);
   end Lire_Contenu_Fichier;
```

Ensuite, nous avons besoin de séparer les différentes parties du contenu compressé. Pour cela, nous avons choisi d'implémenter le sous-programme "Separer_Contenu" qui est chargé de récupérer séparément les caractères, la structure de l'arbre de Huffman, ainsi

que le texte compressé. Le processus commence par l'enregistrement des caractères depuis un tableau dans la liste de type T_LCA fournie, suivi de la construction progressive de la structure de l'arbre de Huffman à partir du flux binaire. Une fois l'arbre construit, le texte compressé est extrait et stocké séparément. C'est le reste du flux binaire.

```
procedure Separer_Contenu(Tableau_Caracteres : in
   T_Liste_Caracteres; Nombre_Caracteres : in Integer;
   Contenu : in Unbounded_String; Liste_Caracteres : out
   LCA_UStr_UStr.T_LCA; Structure_Arbre_Huffman : in out
   Unbounded_String; Texte_Compresse : out Unbounded_String)
   is
        Nombre_Des_Uns : Integer := 0;
        Indice : Integer := 1;
    begin
        for I in 1 .. Nombre_Caracteres loop
            Enregistrer(Liste_Caracteres, Tableau_Caracteres(
               I), To_Unbounded_String(""));
        end loop;

        while Nombre_Des_Uns /= Nombre_Caracteres loop
            if To_String(Contenu)(Indice) = '1' then
                Nombre_Des_Uns := Nombre_Des_Uns + 1;
            end if;
            Structure_Arbre_Huffman :=
               Structure_Arbre_Huffman &
To_Unbounded_String(Character'Image(To_String(Contenu)(Indice
   )));
            Indice := Indice + 1;
        end loop;

        Texte_Compresse := To_Unbounded_String(To_String(
           Contenu)(Indice .. Length(Contenu)));

    end Separer_Contenu;
```

Maintenant, on a besoin de reconstruire l'arbre de Huffman. Le sous-programme "Reconstruire_Arbre_Huffman" a pour objectif de remplir cette tâche à partir de la structure binaire qui a été précédemment enregistrée dans le fichier compressé, et qu'on a stockée dans la variable "Structure_Arbre_Huffman". Il parcourt cette structure et, en fonction des valeurs binaires rencontrées, il crée récursivement les nœuds de l'arbre. Si le bit est '0', on doit faire un passage vers le fils gauche, donc il crée un nœud interne avec des fils gauche et droit et continue de décoder la structure. Si le bit est '1', il crée un nœud feuille avec un caractère associé à partir de la liste de caractères. Le sous-programme "Reconstruire_Arbre_Huffman" fonctionne efficacement pour reconstruire l'arbre de Huffman. L'utilisation des conditions pour distinguer entre les nœuds internes et les nœuds feuilles est appropriée, et l'approche récursive est adaptée à la structure binaire. Cependant, l'usage de la récursion peut entraîner des problèmes de pile si la structure est trop grande, une solution itérative pourrait alors être envisagée pour gérer des structures plus complexes.

```
procedure Reconstruire_Arbre_Huffman(Structure_Arbre_Huffman
   : in out Unbounded_String; Liste_Caracteres : in T_LCA;
   Indice_Caractere : in out Integer; Arbre_Huffman : out
   T_ABR) is
        Arbre: T_ABR;
        Noeud: T_ABR;
        Arbre_null: T_ABR;
        UStr_null : Constant Unbounded_String :=
           To_Unbounded_String("");
    begin
        Initialiser(Arbre_null);
        Initialiser(Arbre);
        if Structure_Arbre_Huffman /= UStr_null then
            Remplacer(Arbre_Huffman, Arbre);
            if To_String(Structure_Arbre_Huffman)(1) = '0'
               then
                Creer_Fils_Gauche(Arbre_Huffman,
                   To_Unbounded_String(""), 0);
                Creer_Fils_Droit(Arbre_Huffman,
                   To_Unbounded_String(""), 0);
                Structure_Arbre_Huffman :=
                   To_Unbounded_String(To_String(
                   Structure_Arbre_Huffman)(2 .. Length(
                   Structure_Arbre_Huffman)));
                Remplacer(Gauche(Arbre_Huffman), Arbre);
                Reconstruire_Arbre_Huffman(
                   Structure_Arbre_Huffman, Liste_Caracteres,
                    Indice_Caractere, Arbre);
                Remplacer(Droit(Arbre_Huffman), Arbre);
                Reconstruire_Arbre_Huffman(
                   Structure_Arbre_Huffman, Liste_Caracteres,
                    Indice_Caractere, Arbre);
            end if;
            if Structure_Arbre_Huffman /= UStr_null then
                if To_String(Structure_Arbre_Huffman)(1) =
                   '1' then
                    Creer_Nouveau_Noeud(Noeud, Cle_Indice(
                       Liste_Caracteres, Indice_Caractere),
                       0, Arbre_null, Arbre_null);
                    Remplacer_Donnees(Noeud, Arbre);
                    Structure_Arbre_Huffman :=
                       To_Unbounded_String(To_String(
                       Structure_Arbre_Huffman)(2 .. Length(
                       Structure_Arbre_Huffman)));
                    Indice_Caractere := Indice_Caractere + 1;
                end if;
            end if;
        end if;
```

```
    end Reconstruire_Arbre_Huffman;
```

Pour la reconstruction de la table de Huffman, comme celle-ci se fait à partir de l'arbre de Huffman qu'on vient d'obtenir, nous avons décidé d'employer la même procédure de construction de la table dans le programme de compression. Le sous-programme, nommé cette fois-ci "Reconstruire_Table_Huffman", a pour but de reconstruire la table de Huffman afin de pouvoir déchiffrer de texte encodé. Il utilise la même procédure interne récursive "Generer_Codes" que précédemment. Les codes sont générés en ajoutant un '0' pour les fils gauches et un '1' pour les fils droits. Lorsque l'arbre atteint une feuille, le code associé au caractère est enregistré dans la table. L'approche récursive permet de traiter l'arbre de manière fluide et de générer les codes binaires de manière directe. Cependant, comme pour la procédure de reconstruction de l'arbre, l'utilisation de la récursion pourrait être problématique si l'arbre est particulièrement grand, risquant de dépasser la limite de profondeur de la pile. Pour un grand arbre, une approche itérative pourrait être envisagée. Dans l'ensemble, ce sous-programme remplit bien sa tâche et est efficace dans la reconstruction de la table de Huffman.

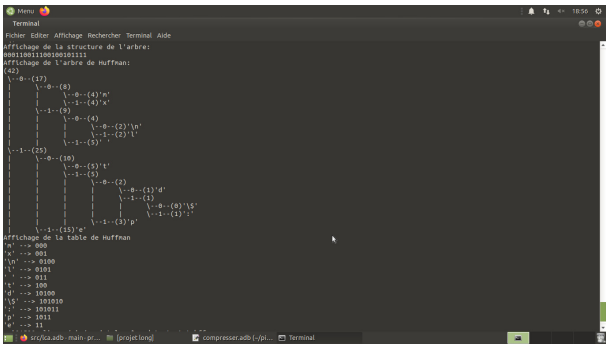
```
procedure Reconstruire_Table_Huffman(Table_Huffman : in out
   LCA_UStr_UStr.T_LCA; Arbre_Huffman : in ABR_UStr_Int.T_ABR
   ) is
        procedure Generer_Codes(Arbre : in ABR_UStr_Int.T_ABR
           ; Code: in Unbounded_String; Table: in out
           LCA_UStr_UStr.T_LCA) is
            Code_G, Code_D : Unbounded_String;
        begin
            if not (Est_Vide(Arbre)) then
                if Est_Feuille(Arbre) then
                    Enregistrer(Table, Symbole(Arbre), Code);
                else
                    Code_G := Code & To_Unbounded_String("0")
                       ;
                    Generer_Codes(Gauche(Arbre), Code_G,
                       Table);
                    Code_D := Code & To_Unbounded_String("1")
                       ;
                    Generer_Codes(Droit(Arbre), Code_D, Table
                       );
                end if;
            end if;
        end Generer_Codes;

    begin
        Initialiser(Table_Huffman);
        Generer_Codes(Arbre_Huffman, Null_Unbounded_String,
           Table_Huffman);
    end Reconstruire_Table_Huffman;
```

Enfin, la procédure "Decoder_Texte" permet la décompression du texte encodé en utilisant la table de Huffman. Elle lit chaque bit du texte compressé, forme progressivement un code binaire, puis cherche ce code dans la table de Huffman. Dès qu'un code est

trouvé, le caractère correspondant est ajouté à la chaîne décompressée, et le processus recommence avec le prochain code. Cette méthode permet de reconstruire le texte original à partir du flux binaire compressé. Après avoir été testé, "Decoder_Texte" fonctionne bien pour décompresser le texte caractère par caractère. L'approche itérative qui forme progressivement les codes binaires à partir du texte compressé est bien adaptée et efficace. L'étape finale est celle de la création et l'écriture du fichier décompressé, qui est simple mais essentielle. Après avoir effectué toutes les étapes précédentes, qui comprennent la reconstruction de l'arbre de Huffman, la génération de la table de Huffman, et le décodage du texte. Cette étape, écrite dans le corps du code de décompression, consiste à créer un fichier décompressé en sortie, où le texte décompressé est écrit dans un nouveau fichier avec l'extension ".d", puis le ferme une fois l'écriture terminée. Cela permet de récupérer un fichier lisible contenant le texte d'origine avant la compression.

2.4 Tests des programmes

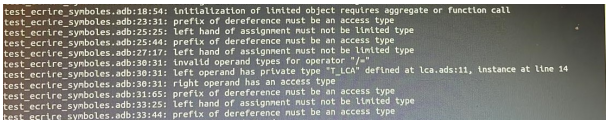
Nous avons utilisé plusieurs programmes de test que nous avons inclus dans le git. Ces programmes nous ont permis de corriger pas à pas les erreurs du code qui s'affichaient et qui entravaient la bonne exécution de nos programmes. Dans le git, nous avons déposé des programmes de test des programmes **compresser** et **décompresser**. Nous avons déposé des tests de la construction de la table de fréquences,l'arbre de Huffman et la table de Huffman pour nous assurer de leur bonne implémentation.
Nous avons aussi testé la fonction Symboles qui permet de construire la table des codes ASCII des caractères présents dans la table de Huffman, tout en respectant la contrainte de la position du symbole "$".
Quant à la décompression, nous avons testé les programmes de lecture du fichier compressé, de reconstruction de l'arbre de Huffman et de la table de Huffman. Enfin , nous avons testé le décodage du texte .
Après avoir réglé les problèmes de compilation, nous avons testé le programme final de compression, nous l'avons d'abord testé pour l'exemple donné dans le sujet et nous avons obtenu des résultats similaires comme suit:



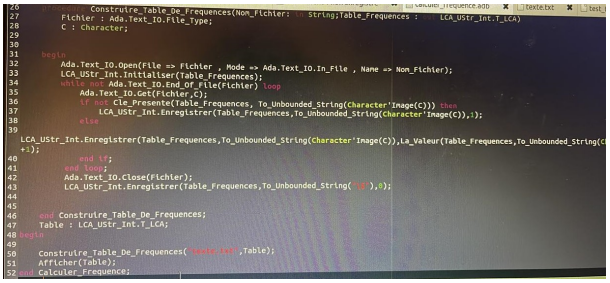
Nous avons fait pareil pour la décompression et nous avons obtenu un fichier avec une extension .d . Nous avons aussi réglé les problèmes des fuites à l'aide de valgrind.

2.5 Difficultés rencontrées

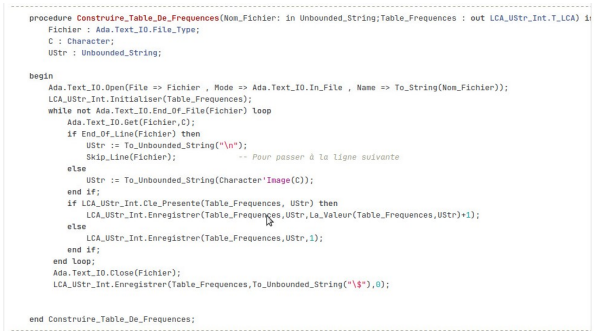
En essayant de compiler tout le programme **compresser**, nous avons remarqué que plusieurs erreurs s'affichaient que nous ne pouvions pas comprendre. Nous avons donc opté pour des programmes de test pour chaque fonction ou procédure phares que nous avons implémentées .
Une erreur qui persistait était :



Nous nous sommes rendues compte que l'utilisation d'un type **limited private** nous interdisait l'affectation de valeurs à l'extérieur du corps du module. Ainsi , nous avons défini les différentes fonctions et procédures mentionnées auparavant, telles que **Valeur,Cle,Symbole,Frequenc** etc., qui nous ont permis de manipuler aisément les types LCA et ABR en dehors du corps des modules.
Un autre problème que nous avons rencontré est l'obtention de la fréquence du retour à la ligne qui ne s'affichait pas dans la table de fréquences avec le programme suivant :
Pour remédier à ce problème, nous avons choisi de traiter le cas du caractère "\n" seul



de la manière suivante :



2.6 Organisation de l'équipe

Au début, nous avons créé les raffinages ensemble. Nous nous sommes mis d'accord sur les types à utiliser comme les listes chaînées associatives et les arbres. Nous avons passé une semaine à peaufiner les raffinages avant de passer aux codes. Nous avons commencé toutes les deux à coder le programme **compresser**. Au fur et à mesure, chacune de nous avait pris la responsabilité de s'occuper d'un programme différent. Aya a terminé le programme **compresser** et Fadwa a commencé le programme **décompresser**. Vu que chacune de nous avait des erreurs soit d'exécution ou de compilation dans son programme, nous nous sommes regroupées encore une fois, et nous nous sommes concentrées d'abord sur le programme **compresser** qui nous a pris beaucoup de temps et qui demandait beaucoup plus d'effort puisque pour nous , la décompression est l'opération inverse de la compression. Ainsi, un programme de compression réussi entraînerait certainement une décompression réussie. Nous sommes parvenues à régler les problèmes de compilation, d'exécution et d'affichage du programme compresser. C'est ici que nous avons attaqué le programme de décompression.
Quant aux fichiers **demo.pdf,presentation.pdf** et **rapport.pdf**, nous avons réparti les tâches entre nous. Fadwa a pris en charge l'intégralité de la présentation et le bilan de la décompression figurant dans le rapport, tandis qu'Aya s'est occupée de la démonstration et du reste du rapport.
Nous donnons ici le tableau qui spécifie les tâches de chacune de nous.

Modules	**Sous-programmes**	**Qui ?**			
		Spécification	**Codage**	**Test**	**Relecture**
Compresser	sp1	Aya/Fadwa	Aya	Aya	Aya/Fadwa
	sp2	Aya	Aya	Aya	Aya/Fadwa
	sp3	Aya/Fadwa	Aya/Fadwa	Aya	Aya/Fadwa
	sp4	Aya/Fadwa	Aya/Fadwa	Aya/Fadwa	Aya/Fadwa
	sp5	Aya/Fadwa	Aya	Aya	Aya/Fadwa
Décompresser	sp1	Aya/Fadwa	Fadwa	Fadwa	Aya/Fadwa
	sp2	Aya/Fadwa	Fadwa	Fadwa	Aya/Fadwa
	sp3	Aya/Fadwa	Fadwa	Fadwa	Aya/Fadwa
	sp4	Aya/Fadwa	Fadwa	Fadwa	Aya/Fadwa

2.7 Bilan du projet

2.7.1 Bilan technique

Nous avons fait la remarque que , faute de temps, nous n'avons pas testé le programme **compresser** par l'option **valgrind** pour voir s'il y a d'éventuelles fuites de mémoire. Nous avons toutefois détruit l'ensemble des structures avec lesquelles nous avons travaillé pour les deux programmes. L'étape de compression nous a pris beaucoup de temps vu les problèmes nombreux que nous avons rencontrés avant l'affinement final du programme. Et même après exécution , nous avons remarqué que le caractère de retour à la ligne ne s'affichait pas dans la table de fréquences. Nous avons alors réglé le problème dans peu de temps, et nous sommes parvenues à obtenir un bon affichage de l'ensemble des structures à l'aide de l'option -b. Cependant, nous trouvons encore un problème avec la lecture du texte compressé avec la commande **xxd** qui ne nous affiche pas la structure désirée et attendue. Nous ne sommes pas parvenues à détecter la source de la faute.
Une autre lacune potentielle que nous avons remarquée dans notre programme est qu'il n'est pas possible de compresser plusieurs fichiers à la fois. Et vu que l'exécution du programme nous a pris beaucoup de temps, nous ne pouvions pas résoudre ce problème. Le programme de compression nous a pris environ deux semaines. Celui de décompression a été réalisé au fur et à mesure que nous finissions le rapport. La chose qui nous a demandé une semaine à peu près.

2.7.2 Bilan individuel

Aya

Le codage de Huffman est une méthode de codage dont j'ai tant entendu parler, vu son efficacité et sa rapidité dans l'execution. J'ai donc trouvé ce projet assez constructif et formateur dans le sens où ce travail m'a permis d'éclaircir davantage ma compréhension de cette méthode de codage et d'enlever les ambiguités.
Malgré l'ensemble des difficultés que ma collègue et moi avons trouvées non pas à comprendre le sujet et son essence, la chose qui était facile pour nous, mais à traduire notre compréhension en un code cohérent et sans erreurs, ce projet était bénéfique pour mettre en oeuvre et en pratique l'ensemble des connaissances et d'astuces acquises lors des séances des TP et des TD.

Fadwa

Ce projet m'a permis d'explorer en profondeur le codage de Huffman à travers la compression et la décompression de fichiers texte. Travailler simultanément sur ces deux programmes a représenté un véritable défi pour moi. La gestion parallèle des deux processus a parfois engendré de la confusion et des erreurs. Cependant, cette difficulté s'est avérée être une opportunité de développement personnel. Elle m'a incité à analyser chaque étape avec rigueur, à approfondir ma compréhension des structures de données utilisées, et à renforcer ma capacité de concentration pour résoudre les problèmes de manière méthodique. L'optimisation progressive de l'algorithme de décompression m'a permis d'améliorer l'efficacité du code tout en renforçant mes compétences en débogage, optimisation et gestion des priorités. Cette expérience a approfondi notre compréhen-

sion des algorithmes tout en renforçant notre rigueur et démarche dans le développement logiciel.

2.8 Conclusion

Ce projet de codage de Huffman a permis de démontrer l'efficacité et la modularité de cette méthode de compression/décompression. Grâce à une approche structurée, nous avons réussi à lire, décoder et reconstruire les données compressées.Nous avons réussi une grande partie de notre projet. Les étapes principales — comme la lecture des fichiers, la reconstruction de l'arbre de Huffman et la décompression des textes — ont été conçues pour garantir la fiabilité du processus. Cependant, il nous reste des erreurs que nous ne sommes malheureusement pas parvenues à corriger .Cette expérience nous a permis d'approfondir nos connaissances sur le codage de Huffman, et nous a donné l'opportunité d'explorer aussi bien les défis que les bienfaits du travail en groupe.