INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Institut National Des Sciences Apliquées

ROUEN NORMANDIE

Rapport de projet

Compresseur d'Huffman

AUTEURS :
THOMAS BAUER
MATHIS SAUNIER
ALI HAMDANI
TAOBA OUATHRANI

Chef de projet : TAOBA OUATHRANI

ITI3 2023-2024

7 Janvier 2024

Table des matières

Chapitre 1

Introduction

1.1 Introdution

La compression de données est un domaine de recherche actif depuis de nombreuses années. Elle vise à réduire la taille des données en conservant le contenu original. La compression est utilisée dans de nombreuses applications, telles que le stockage de données, la transmission de données et le traitement des données.

Parmi les nombreux paradigmes de compression, les deux catégories prédominantes sont les algorithmes de compression par perte et sans perte. Les premiers, sacrifiant une partie de l'information originale pour une compression plus importante, sont souvent utilisés dans des applications où une légère dégradation de la qualité est acceptable. En revanche, les algorithmes sans perte préservent intégralement les données initiales, trouvant leur utilité dans des domaines sensibles à toute altération, tels que les archives numériques, les bases de données et les transmissions sans erreur.

Le présent projet se fixe pour objectif de concevoir un algorithme de compression sans perte, s'appuyant sur la technique du codage de Huffman. Une fois l'algorithme de compression développé, une évaluation de son efficacité sera réalisée par le biais de tests de compression de fichiers de différentes tailles, types et contenus.

La structure du rapport de ce projet suivra un plan comprenant la présentation des Types Abstraits de Données (TADs) et des analyses descendantes, la conception préliminaire, la conception détaillée, l'implémentation du code C et des tests unitaires ainsi qu'une section dédiée à l'organisation du groupe. Enfin, le rapport se conclura par une synthèse globale du projet et des retours sur les résultats obtenus.

Chapitre 2

Types Abstraits de Données

2.1 Analyse : les TAD

2.1.1 TAD Octet

Nom: Octet

Utilise: **Bit**, **0..7**

Opérations:

creerOctet:	**Bit** $\times$ **Bit** $\times$ **Bit** $\times$ **Bit** $\times$ **Bit** $\times$ **Bit** $\times$ **Bit** $\times$ **Bit** $\rightarrow$ **Octet**
obtenirIemeBit:	**Octet** $\times$ **0..7** $\rightarrow$ **Bit**
octetVersNaturel:	**Octet** $\rightarrow$ **0..255**
naturelVersOctet:	**0..255** $\rightarrow$ **Octet**

Axiomes:

- $obtenirIemeBit(creerOctet(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0), 0) = b_0$
  Note : on obtient b_1 pour l'indice 1, ... et b_7 pour l'indice 7

Sémantiques:

octetVersNaturel:	permet la conversion d'un nombre en base 2 (Octet) vers un nombre en base 10 (Naturel)
naturelVersOctet:	permet la conversion d'un nombre en base 10 (Naturel) vers un nombre en base 2 (Octet)

2.1.2 TAD Statistiques

Nom: Statistiques

Utilise: **Octet**

Opérations:

statistiques:	$\rightarrow$ **Statistiques**
incrementerOccurrence:	**Statistiques** $\times$ **Octet** $\rightarrow$ **Statistiques**
obtenirOccurrence:	**Statistiques** $\times$ **Octet** $\rightarrow$ **Naturel**
fixerOccurrence:	**Statistiques** $\times$ **Octet** $\times$ **Naturel** $\rightarrow$ **Statistiques**

Axiomes:

- $obtenirOccurrence(statistique(), o) = 0$
- $obtenirOccurrence(incrementerOccurrence(s, o), o) = obtenirOccurrence(s, o) + 1$
- $obtenirOccurrence(fixerOccurrence(s, o, n), o) = n$

2.1.3 TAD FileDePriorite

Nom: FileDePriorite

Paramètre: Element ($\forall e_1 \in Element, e_2 \in Element, e_1 \neq e_2 \Rightarrow e_1 < e_2$ *ou* $e_1 > e_2$)

Utilise: **Booleen**

Opérations:

fileDePriorite:	$\rightarrow$ **FileDePriorite**
estVide:	**FileDePriorite** $\rightarrow$ **Booleen**
enfiler:	**FileDePriorite** $\times$ **Element** $\rightarrow$ **FileDePriorite**
obtenirElementEtDefiler:	**FileDePriorite** $\nrightarrow$ **FileDePriorite** $\times$ **Element**

Préconditions: obtenirElementEtDefiler(f): non(estVide(f))

Axiomes:

- $estVide(fileDePriorite())$
- $non(estVide(enfiler(f, e)))$
- $obtenirElementEtDefiler(enfiler(fileDePriorite(), e)) = fileDePriorite(), e$
- $e \leq obtenirElementEtDefiler(f)[2] \Rightarrow obtenirElementEtDefiler(enfiler(f, e)) = f, e$
- $e > obtenirElementEtDefiler(f)[2] \Rightarrow obtenirElementEtDefiler(enfiler(f, e)) = enfiler(obtenirElementEtDefiler(f)[1], e), obtenirElementEtDefiler(f)[2]$

2.1.4 TAD ArbreDeHuffman

Nom: **ArbreDeHuffman**

Utilise: **Octet**, **Naturel**, **Booleen**

Opérations:

arbreDeHuffman:	**Octet** $\times$ **Naturel** $\rightarrow$ **ArbreDeHuffman**
fusionner:	**ArbreDeHuffman** $\times$ **ArbreDeHuffman** $\rightarrow$ **ArbreDeHuffman**
estUneFeuille:	**ArbreDeHuffman** $\rightarrow$ **Booleen**
obtenirOctet:	**ArbreDeHuffman** $\nrightarrow$ **Octet**
obtenirFrequence:	**ArbreDeHuffman** $\rightarrow$ **Naturel**
obtenirFilsGauche:	**ArbreDeHuffman** $\nrightarrow$ **ArbreDeHuffman**
obtenirFilsDroit:	**ArbreDeHuffman** $\nrightarrow$ **ArbreDeHuffman**

Préconditions:

obtenirOctet(a):	estUneFeuille(a)
obtenirFilsGauche(a):	non(estUneFeuille(a))
obtenirFilsDroit(a):	non(estUneFeuille(a))

Axiomes:

- $estUneFeuille(arbreDeHuffman(o, f))$
- $non(estUneFeuille(fusionner(a_g, a_d)))$
- $obtenirOctet(arbreDeHuffman(o, f)) = o$
- $obtenirFrequence(arbreDeHuffman(o, f)) = f$
- $obtenirFrequence(fusionner(a_g, a_d)) = obtenirFrequence(a_g) + obtenirFrequence(a_d)$
- $obtenirFilsGauche(fusionner(a_g, a_d)) = a_g$
- $obtenirFilsDroit(fusionner(a_g, a_d)) = a_d$

2.1.5 TAD CodeBinaire

Nom: CodeBinaire

Utilise: **Octet**, **Naturel**, **Bit**

Opérations:
- creerCodeBinaire: **Bit** $\rightarrow$ **CodeBinaire**
- ajouterBit: **CodeBinaire** $\times$ **Bit** $\rightarrow$ **CodeBinaire**
- obtenirIemeBit: **CodeBinaire** $\times$ **Naturel** $\nrightarrow$ **Bit**
- obtenirLongueur: **CodeBinaire** $\rightarrow$ **Naturel**

Préconditions:
- obtenirIemeBit(cb, i): i < obtenirLongueur(cb)

Axiomes:
- $obtenirLongueur(creerCodeBinaire(b)) = 1$
- $obtenirLongueur(ajouterBit(cb, b)) = obtenirLongueur(cb) + 1$

Sémantiques:
- ajouterBit: Ajoute le bit en question à la fin du CodeBinaire
- obtenirIemeBit: Retourne le bit à la position i du CodeBinaire (0 étant la position du premier bit)

2.1.6 TAD TableDeCodage

Nom: TableDeCodage

Utilise: **Octet**, **Booleen**, **CodeBinaire**

Opérations:
- creerTableCodage: $\rightarrow$ **TableDeCodage**
- ajouterCodage: **TableDeCodage** $\times$ **Octet** $\times$ **CodeBinaire** $\nrightarrow$ **TableDeCodage**
- octetPresent: **TableDeCodage** $\times$ **Octet** $\rightarrow$ **Booleen**
- octetVersCodeBinaire: **TableDeCodage** $\times$ **Octet** $\nrightarrow$ **CodeBinaire**

Préconditions:
- ajouterCodage(t,octet,codeBinaire): non(octetPresent(t, octet))
- octetVersCodeBinaire(t, octet): octetPresent(t, octet)

Axiomes:
- $octetPresent(ajouterCodage(t, octet, codeBinaire), octet)$
- $non(octetPresent(creerTableCodage(), octet))$
- $octetVersCodeBinaire(ajouterCodage(t, octet, codeBinaire), octet) = codeBinaire$

2.2 Conception Préliminaire

2.2.1 Signatures et fonctions de Octet

fonction creerOctet **(**b7, b6, b5, b4, b3, b2, b1, b0 : **Bit) : Octet**
fonction obtenirIemeBit **(**o : **Octet**, i : **0..7) : Bit**
fonction octetVersNaturel **(**o : **Octet) : 0..255**
fonction naturelVersOctet **(**n : **0..255) : Octet**

2.2.2 Signatures et fonctions de Statistiques

fonction statistiques **() : Statistiques**
procédure incrementerOccurrence **(E/S** s : **Statistiques**, **E** o : **Octet)**
fonction obtenirOccurrence **(**s : **Statistiques**, o : **Octet) : Naturel**
procédure fixerOccurrence **(E/S** s : **Statistiques**, **E** o : **Octet**, n : **Naturel)**

2.2.3 Signatures et fonctions de FileDePriorite

fonction fileDePriorite **() : FileDePriorite**
fonction estVide (fdp : **FileDePriorite) : Booleen**
procédure enfiler **(E/S** fdp : **FileDePriorite**, **E** e : **Element)**
procédure obtenirElementEtDefiler **(E/S** fdp : **FileDePriorite**, **S** e : **Element)**

⌊**précondition(s)** non(estVide(fdp))

2.2.4 Signatures et fonctions de ArbreDeHuffman

fonction arbreDeHuffman **(**o : **Octet**, n : **Naturel) : ArbreDeHuffman**
fonction fusionner **(**ag : **ArbreDeHuffman**, ad : **ArbreDeHuffman) : ArbreDeHuffman**
fonction estUneFeuille **(**a : **ArbreDeHuffman) : Booleen**
fonction obtenirOctect **(**a : **ArbreDeHuffman) : Octet**

⌊**précondition(s)** estUneFeuille(a)

fonction obtenirFrequence **(**a : **ArbreDeHuffman) : Naturel**
fonction obtenirFilsGauche **(**a : **ArbreDeHuffman) : ArbreDeHuffman**

⌊**précondition(s)** non(estUneFeuille(a))

fonction obtenirFilsDroit **(**a : **ArbreDeHuffman) : ArbreDeHuffman**

⌊**précondition(s)** non(estUneFeuille(a))

procédure liberer **(E** arbre : **ArbreDeHuffman)**
// Procédure métier permettant de libérer un arbre de Huffman de la mémoire

2.2.5 Signatures et fonctions de CodeBinaire

fonction creerCodeBinaire **(**b : **Bit) : CodeBinaire**
procédure ajouterBit **(E/S** cb : **CodeBinaire**, **E** b : **Bit)**
fonction obtenirIemeBit **(**cb : **CodeBinaire**, i : **Naturel) : Bit**

⌊**précondition(s)** i < obtenirLongueur(cb)

fonction obtenirLongueur **(**cb : **CodeBinaire) : Naturel**

2.2.6 Signatures et fonctions de TableDeCodage

fonction creerTableCodage **() : TableDeCodage**
procédure ajouterCodage **(E/S** tdc : **TableDeCodage**, **E** o : **Octet**, cb : **CodeBinaire)**
fonction OctetPresent **(**tdc : **TableDeCodage**, o : **Octet) : Booleen**
fonction OctetVersCodeBinaire **(**tdc : **TableDeCodage**, o : **Octet) : CodeBinaire**

⌊**précondition(s)** octetPresent(t, octet)

2.3 Conception Détaillée

// Dans le but de simplifier la conception détaillée ainsi que le développement, nous considérons bitA0 = 0 et bitA1 = 1. Nous pouvons ainsi utiliser le type Bit comme un naturel. Cela permet d'éviter des instructions conditionnelles répétitives et coûteuses bien que triviales.

2.3.1 Algorithmes de Octet

Type Octet = **0..255**

fonction creerOctet **(**b7, b6, b5, b4, b3, b2, b1, b0 : **Bit) : Octet**

Déclaration o : **Octet**

debut
o ← b0 + b1*2 + b2*2^2 + b3*2^3 + b4*2^4 + b5*2^5 + b6*2^6 + b7*2^7
retourner o
fin

fonction obtenirIemeBit **(**o : **Octet**, b : **0..7) : Bit**
debut
retourner (o **div** 2^b) **mod** 2
fin

fonction octetVersNaturel **(**o : **Octet) : 0..255**
debut
retourner o
fin

fonction naturelVersOctet **(**n : **0..255) : Octet**
debut
retourner n
fin

2.3.2 Algorithmes de Statistiques

Type Statistiques = **Tableau[**0..255**] de Naturel**

fonction statistiques **() : Statistiques**

Déclaration s : **Statistiques**

debut
pour octet ← 0 **à** 255 **faire**
s[octet] ← 0
finpour
retourner s
fin

procédure incrementerOccurrence **(E/S** s : **Statistiques**, **E** o : **Octet)**
debut
s[octetVersNaturel(o)] ← s[octetVersNaturel(o)] + 1
fin

fonction obtenirOccurrence **(**s : **Statistiques**, o : **Octet) : Naturel**
debut
  retourner s[octetVersNaturel(o)]
fin
procédure fixerOccurrence **(E/S** s : **Statistiques**, **E** o : **Octet**, n : **Naturel)**
debut
  s[octetVersNaturel(o)] ← n
fin

2.3.3 Algorithmes de FileDePriorite

procédure enfiler **(E/S** fdp : **FileDePriorite**, **E** e : **Element)**
  Déclaration temp : **FileDePriorite**
debut
  si estVide(fdp) ou e < fdp^.element **alors**
    allouer(temp)
    temp^.element ← e
    temp^.fileSuivante ← fdp
    fdp ← temp
  sinon
    enfiler(fdp^.fileSuivante, e)
  finsi
fin

procédure obtenirElementEtDefiler **(E/S** fdp : **FileDePriorite**, **S** e : **Element)**
  ⌊**précondition(s)** non(estVide(fdp))
  Déclaration temp : **FileDePriorite**
debut
  e ← fdp^.element
  temp ← fdp
  fdp ← temp^.listeSuivante
  desallouer(temp)
fin

fonction fileDePriorite **() : FileDePriorite**
debut
  retourner NIL
fin

fonction estVide **(**fdp : **FileDePriorite) : Booleen**
debut
  retourner fdp = **NIL**
fin

2.3.4 Algorithmes de ArbreDeHuffman

Type ArbreDeHuffman =^Noeud
Type Noeud = **Structure**

```
    octet : Octet
    frequence : Naturel
    estUneFeuille : Booleen
    filsGauche : ArbreDeHuffman
    filsDroit : ArbreDeHuffman
finstructure

fonction arbreDeHuffman (o : Octet, f : Naturel) : ArbreDeHuffman
    Déclaration   a : ArbreDeHuffman
debut
    allouer(a)
    a^.octet ← o
    a^.frequence ← f
    a^.estUneFeuille ← Vrai
    a^.filsGauche ← NIL
    a^.filsDroit ← NIL
    retourner a
fin

fonction fusionner (a_g, a_d : ArbreDeHuffman) : ArbreDeHuffman
    Déclaration   racine : ArbreDeHuffman
debut
    allouer(racine)
    racine^.filsGauche ← a_g
    racine^.filsDroit ← a_d
    racine^.estUneFeuille ← Faux
    racine^.frequence ← obtenirFrequence(a_g) + obtenirFrequence(a_d)
    retourner racine
fin

fonction estUneFeuille (a : ArbreDeHuffman) : Booleen
debut
    retourner a^.estUneFeuille
fin

fonction obtenirOctet (a : ArbreDeHuffman) : Octet
    ⌊précondition(s)   estUneFeuille(a)
debut
    retourner a^.octet
fin

fonction obtenirFrequence (a : ArbreDeHuffman) : Naturel
debut
    retourner a^.frequence
fin

fonction obtenirFilsGauche (a : ArbreDeHuffman) : ArbreDeHuffman
    ⌊précondition(s)   non(estUneFeuille(a))
debut
    retourner a^.filsGauche
```

fin

fonction obtenirFilsDroit **(**a : **ArbreDeHuffman) : ArbreDeHuffman**
  ⌊**précondition(s)** non(estUneFeuille(a))
debut
  retourner a^.filsDroit
fin

procédure liberer **(E** arbre : **ArbreDeHuffman)**
debut
  si non(estUneFeuille(arbre)) **alors**
    liberer(obtenirFilsGauche(arbre))
    liberer(obtenirFilsDroit(arbre))
  finsi
  desallouer(arbre)
fin

2.3.5 Algorithmes de CodeBinaire

Type CodeBinaire = **Structure**
  codeBinaire : **Naturel**
  nbBits : **Naturel**
finstructure

fonction creerCodeBinaire **(**b : **Bit) : CodeBinaire**
  Déclaration cb : **CodeBinaire**
debut
  cb.codeBinaire ← b
  cb.nbBits ← 1
  retourner cb
fin

fonction obtenirIemeBit **(**cb : **CodeBinaire**, i : **Naturel) : Bit**
  ⌊**précondition(s)** i < cb.nbBits
debut
  retourner (cb.codeBinaire **div** 2^i) **mod** 2
fin

fonction obtenirLongueur **(**cb : **CodeBinaire) : Naturel**
debut
  retourner cb.nbBits
fin

procédure ajouterBit **(E/S** cb : **CodeBinaire**, **E** b : bit**)**
debut
  cb.codebinaire ← cb.codebinaire + b * 2^(cb.nbBits)
  cb.nbBits ← cb.nbBits + 1
fin

2.3.6 Algorithmes de TableDeCodage

Type TableDeCodage = **Structure**
  tableDeCodeBinaire : **Tableau[**0..255**] de CodeBinaire**
  tableDePresence : **Tableau[**0..255**] de Booleen**
finstructure

fonction creerTableCodage **() : TableDeCodage**
  Déclaration tdc : **TableDeCodage**
debut
  pour octet ← 0 **à** 255 **faire**
    tdc.tableDePresence[octet] ← **Faux**
  finpour
  retourner tdc
fin

procédure ajouterCodage **(E/S** tdc : **TableDeCodage**, **E** o : **Octet**, cb : **CodeBinaire)**
  ⌊**précondition(s)** non(octetPresent(t, octet))
debut
  octet ← octetVersNaturel(o)
  tdc.tableDeCodeBinaire[octet] ← cb
  tdc.tableDePresence[octet] ← **Vrai**
fin

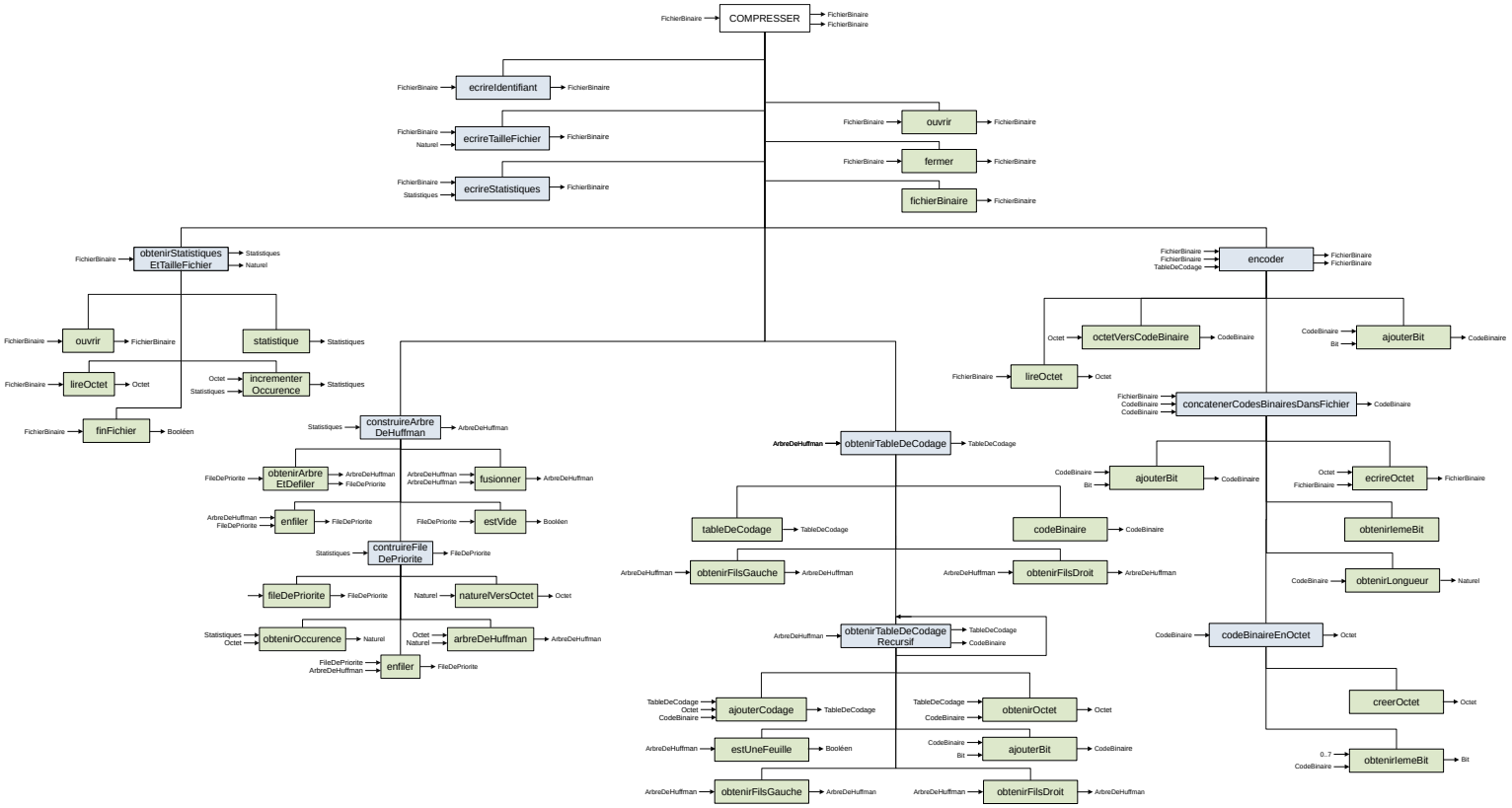
fonction octetVersCodeBinaire **(**tdc : **TableDeCodage**, o : **Octet) : CodeBinaire**
  ⌊**précondition(s)** octetPresent(t, octet)
debut
  retourner tdc.tableDeCodeBinaire[octetVersNaturel(o)]
fin

fonction octetPresent **(**tdc : **TableDeCodage**, o : **Octet) : Booleen**
debut
  retourner tdc.tableDePresence[octetVersNaturel(o)]
fin

Chapitre 3

Compression

3.1 Analyse descendante



3.2 Conception Préliminaire

procédure obtenirStatistiquesEtTailleFichier **(E/S** f : **FichierBinaire**, **S** s : **Statistiques**, taille : **Naturel)**

⌊**précondition(s)** estOuvert(f) et (mode(f) = lecture)

fonction construireFileDePriorite **(**s : **Statistiques) : FileDePriorite**

fonction construireArbreDeHufmman **(**s : **Statistiques) : ArbreDeHuffman**

procédure obtenirTableDeCodageRecursif **(E/S** tdc : **TableDeCodage**, **E** a : **ArbreDeHuffman**, cb : **CodeBinaire)**
fonction obtenirTableDeCodage **(**a : **ArbreDeHuffman) : TableDeCodage**

⌊**précondition(s)** non(estUneFeuille(a))

procédure ecrireIdentifiant **(E/S** fb : **FichierBinaire)**

⌊**précondition(s)** estOuvert(fb) et (mode(fb) = écriture)

procédure ecrireTailleFichier **(E/S** fb : **FichierBinaire**, **E** taillefb : **Naturel)**

⌊**précondition(s)** estOuvert(fb) et (mode(fb) = écriture)

procédure ecrireStatistiques **(E/S** fb : **FichierBinaire**, **E** s : **Statistiques)**

⌊**précondition(s)** estOuvert(fb) et (mode(fb) = écriture)

fonction codeBinaireEnOctet **(**cb : **CodeBinaire) : Octet**

⌊**précondition(s)** obtenirLongueur(cb) = 8

procédure concatenerCodeBinaireDansFichier **(E/S** fbCompresse : **FichierBinaire**, cbtemp : **CodeBinaire**, **E** cb : **CodeBinaire)**

⌊**précondition(s)** estOuvert(fbCompresse) et (mode(fbCompresse) = écriture)

procédure encoder **(E/S** fbInitial : **FichierBinaire**, fbCompresse : **FichierBinaire**, **E** tdc : **TableDeCodage)**

⌊**précondition(s)** estOuvert(fb1) et (mode(fb1) = lecture) et estOuvert(fb2) et (mode(fb2) = écriture)

procédure compresser **(E/S** f : **FichierBinaire**, **S** fCompresse : **FichierBinaire)**

⌊**précondition(s)** estOuvert(f) et (mode(f) = lecture)

3.3 Conception Détaillée

procédure obtenirStatistiquesEtTailleFichier **(E/S** f : **FichierBinaire**, **S** s : **Statistiques**, taille : **Naturel)**

⌊**précondition(s)** estOuvert(f) et (mode(f) = lecture)

Déclaration s : **Statistiques**, taille : Naturel

debut
positionnerAuDebut(f)
s ← statistiques()
taille ← 0
tant que non(finFichier(f)) **faire**
incrementerOccurence(s, lireOctet(f))
taille ← taille + 1
fintantque
fin

fonction construireFileDePriorite **(**s : **Statistiques) : FileDePriorite**

Déclaration fdp : **FileDePriorite**, octet : **Octet**, occurence, o : **Naturel**

debut
fdp ← fileDePriorite()
pour o ← 0 **à** 255 **faire**
octet ← naturelVersOctet(o)
occurence ← obtenirOccurence(s, octet)
si occurence > 0 **alors**
enfiler(fdp, arbreDeHuffman(octet, occurence))
finsi
finpour
retourner fdp
fin

fonction construireArbreDeHuffman **(**s : **Statistiques) : ArbreDeHuffman**

Déclaration fdp : **FileDePriorite**, dernierElement : **Booleen**, a1, a2, aFusion : **ArbreDeHuffman**

debut
fdp ← construireFileDePriorite(s)
dernierElement ← **Faux**
tant que non(dernierElement) **faire**
obtenirElementEtDefiler(fdp, a1)
si estVide(fdp) **alors**
dernierElement ← **Vrai**
sinon
obtenirElementEtDefiler(fdp, a2)
aFusion ← fusionner(a1, a2)
enfiler(fdp, aFusion)
finsi
fintantque
retourner a1
fin

procédure obtenirTableDeCodageRecursif **(E/S** tdc : **TableDeCodage**, **E** a : **ArbreDeHuffman**, cb : **CodeBinaire)**

```
    Déclaration  cbCopie : CodeBinaire
debut
    si estUneFeuille(a) alors
        ajouterCodage(tdc, obtenirOctet(a), cb)
    sinon
        cbCopie ← cb
        ajouterBit(cbCopie, bitA0)
        obtenirTableDeCodageRecursif(tdc, obtenirFilsGauche(a), cbCopie)
        ajouterBit(cb, bitA1)
        obtenirTableDeCodageRecursif(tdc, obtenirFilsDroit(a), cb)
    finsi
fin
fonction obtenirTableDeCodage (a : ArbreDeHuffman) : TableDeCodage
    ⌊précondition(s)  non(estUneFeuille(a))
    Déclaration  tdc : TableDeCodage, cbGauche, cbDroit : CodeBinaire
debut
    tdc ← creerTableDeCodage()

    cbGauche ← creerCodeBinaire(bitA0)
    cbDroit ← creerCodeBinaire(bitA1)
    obtenirTableDeCodageRecursif(tdc, obtenirFilsGauche(a), cbGauche)
    obtenirTableDeCodageRecursif(tdc, obtenirFilsDroit(a), cbDroit)

    retourner tdc
fin

procédure ecrireIdentifiant (E/S fb : FichierBinaire)
    ⌊précondition(s)  estOuvert(fb) et (mode(fb) = écriture)
    Déclaration  id : Naturel
debut
    id ← 1000
    ecrireNaturel(fb,id)
fin
procédure ecrireTailleFichier (E/S fb : FichierBinaire,E taillefb : Naturel)
    ⌊précondition(s)  estOuvert(fb) et (mode(fb) = écriture)
debut
    ecrireNaturel(fb, taillefb)
fin
procédure ecrireStatistiques (E/S fb : FichierBinaire, E s : Statistiques)
    ⌊précondition(s)  estOuvert(fb) et (mode(fb) = écriture)
    Déclaration  o, occurence : Naturel, octet : Octet
debut
    pour o ← 0 à 255 faire
        octet ← naturelVersOctet(o)
        occurence ← obtenirOccurence(s, octet)
        si occurence > 0 alors
            ecrireNaturel(fb, occurence)
            ecrireOctet(fb, octet)
```

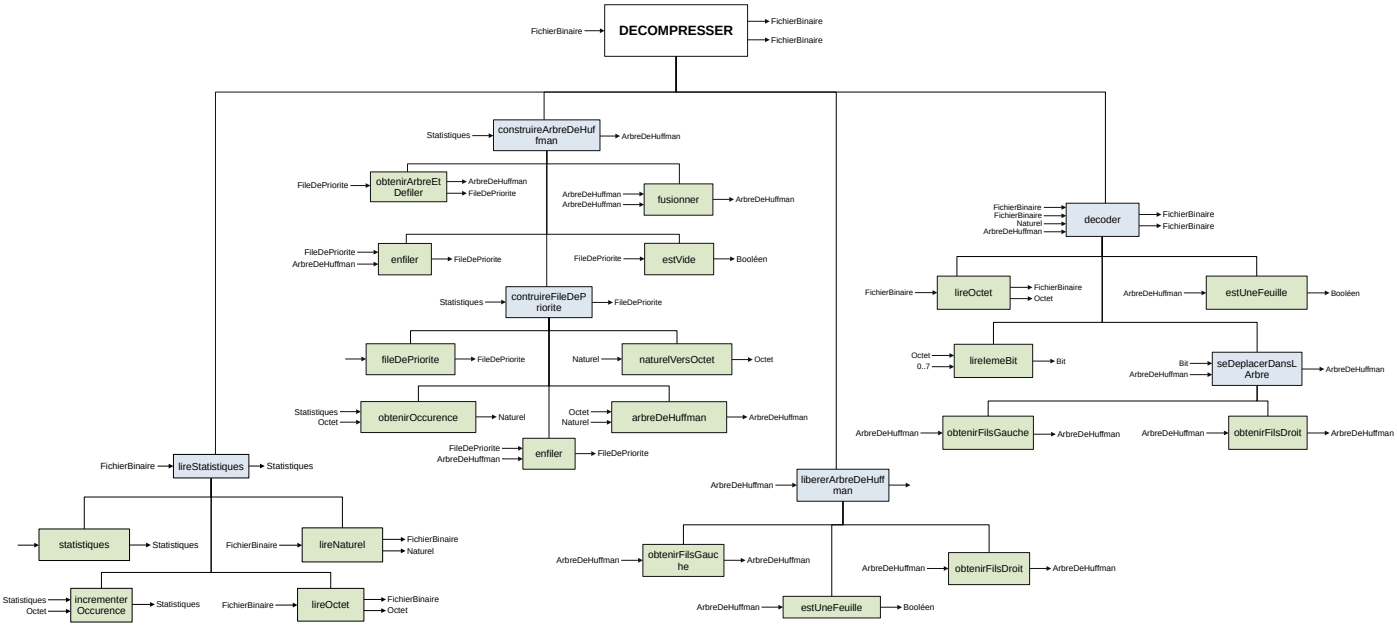
```
        finsi
    finpour
    // On écrit finalement une occurence nulle comme indicateur de fin de lecture de statistiques.
    ecrireNaturel(fb, 0)
fin

fonction codeBinaireEnOctet (cb : FichierBinaire) : Octet
    ⌊précondition(s)  obtenirLongueur(cb) = 8
debut
    retourner creerOctet(obtenirIemeBit(cb,7), obtenirIemeBit(cb, 6), obtenirIemeBit(cb, 5), obtenirIemeBit(cb, 4), obtenirIemeBit(cb, 3), obtenirIemeBit(cb, 2), obtenirIemeBit(cb, 1), obtenirIemeBit(cb, 0))
fin

procédure concatenerCodeBinaireDansFichier (E/S fbCompresse : FichierBinaire, cbtemp : CodeBinaire,E cb : CodeBinaire)
    ⌊précondition(s)  estOuvert(fbCompresse) et (mode(fbCompresse) = écriture)
    Déclaration  i, tailleCb : Naturel
debut
    i ← 0
    tailleCb ← obtenirLongueur(cb)

    tant que i ≠ tailleCb faire
        si obtenirLongueur(cbTemp) = 8 alors
            cbTemp ← creerCodeBinaire(obtenirIemeBit(cb, i))
            i ← i + 1
        finsi

        tant que i < tailleCb et obtenirLongueur(cbTemp) < 8 faire
            ajouterBit(cbTemp, obtenirIemeBit(cb, i))
            i ← i + 1
        fintantque

        si obtenirLongueur(cbTemp) = 8 alors
            ecrireOctet(fbCompresse, codeBinaireEnOctet(cbtemp))
        finsi
    fintantque
fin

procédure encoder (E/S fbInitial : FichierBinaire, fbCompresse : FichierBinaire, E tdc : TableDeCodage)
    ⌊précondition(s)  estOuvert(fb1) et (mode(fb1) = lecture) et estOuvert(fb2) et (mode(fb2) = écriture)
    Déclaration  cbTemp, cb : CodeBinaire, o : Octet, i : Naturel
debut
    positionnerAuDebut(f)

    cbTemp ← creerCodeBinaire(bitA0)
    pour i ← 1 à 7 faire
        ajouterBit(cbTemp, bitA0)
```

```
    finpour

    tant que non finfichier(fnInitiale) faire
        lireoctet(fbInitiale, o)
        cb ← octetVersCodeBinaire(tdc, o)
        concatenerCodeBinaireDansFichier(cbTemp, cb, fbcompresse)
    fintantque

    si obtenirLongueur(cbTemp) < 8 alors
        pour i ← obtenirLongeur(cbTemp) à 7 faire
            ajouterBit(cbTemp, bitA0)
        finpour
        ecrireOctet(fbCompresse, codeBinaireEnOctet(cbTemp))
    finsi
fin

procédure compresser (E/S f : FichierBinaire, S fCompresse : FichierBinaire)
    ⌊précondition(s)  estOuvert(f) et (mode(f) = lecture)
    Déclaration  s : Statistiques, taillefb : Naturel
debut
    positionnerAuDebut(f)
    ouvrir(fbCompresse, ecriture)

    obtenirStatistiquesEtTailleFichier(f, s, taillefb)

    ecrireIdentifiant(fbCompresse)
    ecrireTailleFichier(fbCompresse, taillefb)
    si taillefb > 0 alors
        ecrireStatistiques(fbCompresse, s)
        a ← construireArbreDeHuffman(s)
        si non(estUneFeuille(a)) alors
            encoder(fb, obtenirTableDeCodage(a), fCompresse)
        finsi
        liberer(a)
    finsi

    fermer(fbCompresse)
fin
```

Chapitre 4

Décompression

4.1 Analyse descendante



4.2 Conception Préliminaire

fonction lireStatistiques **(**fb : **FichierBinaire) : Statistiques**

⌊**précondition(s)** estOuvert(fb) et (mode(fb) = lecture)

fonction construireFileDePriorite **(**s : **Statistiques) : FileDePriorite**

fonction construireArbreDeHufmman **(**s : **Statistiques) : ArbreDeHuffman**

procédure seDeplacerDansLArbre **(E** bit : **Bit**, **E/S** arbreCourant : **ArbreDeHuffman)**

procédure decoder **(E** aHuff : **ArbreDeHuffman**, longueur : **Naturel**, **E/S** fb1 : **FichierBinaire**, fb2 : **FichierBinaire)**

⌊**précondition(s)** estOuvert(fb1) et (mode(fb1) = lecture) et estOuvert(fb2) et (mode(fb2) = écriture)

procédure decompresser **(E/S** fCompresse : **FichierBinaire**, **S** fDecompresse : **FichierBinaire)**

⌊**précondition(s)** estOuvert(fCompresse) et (mode(fCompresse) = lecture)

4.3 Conception Détaillée

fonction lireStatistiques **(**fb : **FichierBinaire) : Statistiques**

⌊**précondition(s)** estOuvert(fb) et (mode(fb) = lecture)

Déclaration s : **Statistiques**, octet : **Octet**, occurrence : **Naturel**

debut
s ← statistiques()
// La fonction lireStatistiques est utilisée quand le curseur est au bon endroit et l'on peut donc directement lire les statistiques qui sont des naturels
repeter
lireNaturel(fb, occurrence)
si (occurrence ! = 0) **alors**
lireOctet(fb, octet)
fixerOccurrence(s, octet, occurrence)
finsi
jusqu'a ce que occurrence = 0
retourner s
fin

fonction construireFileDePriorite **(**s : **Statistiques) : FileDePriorite**

Déclaration fdp : **FileDePriorite**, octet : **Octet**, occurence, o : **Naturel**

debut
fdp ← fileDePriorite()
pour o ← 0 **à** 255 **faire**
octet ← naturelVersOctet(o)
occurence ← obtenirOccurence(s, octet)
si occurence > 0 **alors**
enfiler(fdp, arbreDeHuffman(octet, occurence))
finsi
finpour
retourner fdp
fin

fonction construireArbreDeHuffman **(**s : **Statistiques) : ArbreDeHuffman**

Déclaration fdp : **FileDePriorite**, dernierElement : **Booleen**, a1, a2, aFusion : **ArbreDeHuffman**

debut
fdp ← construireFileDePriorite(s)
dernierElement ← **Faux**
tant que non(dernierElement) **faire**
obtenirElementEtDefiler(fdp, a1)
si estVide(fdp) **alors**
dernierElement ← **Vrai**
sinon
obtenirElementEtDefiler(fdp, a2)
aFusion ← fusionner(a1, a2)
enfiler(fdp, aFusion)
finsi
fintantque
retourner a1
fin

```
procédure seDeplacerDansLArbre (E bit : Bit, E/S arbreCourant : ArbreDeHuffman)
debut
    si bit = bitA0 alors
        arbreCourant ← obtenirFilsGauche(arbreCourant)
    sinon
        arbreCourant ← obtenirFilsDroit(arbreCourant)
    finsi
fin
```

```
procédure decoder (E aHuff : ArbreDeHuffman, longueur : Naturel, E/S fb1 : FichierBinaire, fb2 : FichierBinaire)
    ⌊précondition(s)  estOuvert(fb1) et (mode(fb1) = lecture) et estOuvert(fb2) et (mode(fb2) = écriture)
    Déclaration  aTemp : ArbreDeHuffman, finDecodage : Booleen, octetCourant, octetDecodé : Octet, bit : Bit, compteurOctetsDecodes, i : Naturel
debut
    positionnerAuDebut(fb2)
    aTemp ← aHuff
    compteurOctetsDecodes ← 0
    finDecodage ← Faux
    tant que non(finDecodage) faire
        octetCourant ← lireOctet(fb1)
        pour i ←0 à 7 faire
            si non(finDecodage) alors
                bit ← ObtenirIemebit(octetCourant, i)
                seDeplacerDansArbre(aTemp, bit)
                si estUneFeuille(aTemp) alors
                    octetDecode ← obtenirOctet(arbreCourant)
                    ecrireOctet(octetDecodé, fb2)
                    aTemp ← aHuff
                    compteurOctetsDecodes ← compteurOctetsDecodes + 1
                    finDecodage ← (compteurOctetsDecodes = longueur)
                finsi
            finsi
        finpour
    fintantque
fin
```

```
procédure decompresser (E/S fCompresse : FichierBinaire, S fDecompresse : FichierBinaire)
    ⌊précondition(s)  estOuvert(fCompresse) et (mode(fCompresse) = lecture)
    Déclaration  i, id, longueur : Naturel, s : Statistiques, abh : ArbreDeHuffman, octetUnique : Octet
debut
    positionnerAuDebut(fCompresse)
    ouvrir(fDecompresse, ecriture)
    lireNaturel(fCompresser, id)
    // On vérifie si on retrouve bien notre identifiant (ici 1000 sous sa forme de naturel), si ce n'est pas le cas, on n'a pas a décompresser le fichier
    si id = 1000 alors
```

```
        lireNaturel(fCompresser, longueur)
        // Cas particulier d'un fichier vide
        si (longueur > 0) alors
            lireStatistiques(fCompresser, s)
            construireArbreDeHuffman(s, abh)
            // Cas particulier d'un fichier contenant un seul octet (présent plusieurs fois ou non)
            si estUneFeuille(abh) alors
                octetUnique ← octetVersNaturel(obtenirOctet(abh))
                pour i ←1 à longueur faire
                    ecrireOctet(fDecompresse, octetUnique)
                finpour
            sinon
                decoder(abh, longueur, fCompresse, fDecompresse)
            finsi
            liberer(abh)
        finsi
    finsi
    fermer(fDecompresse)
fin
```

Chapitre 5

Développement

5.1 Fichiers d'en-tête

5.1.1 TAD Octet

```
/**
 * \file codeBinaire.h
 * \brief Implémentation du TAD octet pour le compresseur d'Huffman
 * \author A. Hamdani
 * \date 06/01/2023
 *
 */
#ifndef __OCTET__
#define __OCTET__

#include <stdbool.h>

#define MAX_BITS    8
#define bitA0       0
#define bitA1       1
#define MAX_OCTET   256

/**
 * \brief Le type O_Bit est un booléen pouvant prendre la valeur bitA0 (0) ou bitA1 (1)
 *
 */
typedef bool O_Bit;

/**
 * \brief Le type O_Octet est un naturel pouvant prendre des valeurs comprises entre 0 et 255
 ↪   (soit 8 bits)
 *
 */
typedef unsigned char O_Octet;

/**
 * \fn O_Octet O_creerOctet(O_Bit b7, O_Bit b6, O_Bit b5, O_Bit b4, O_Bit b3, O_Bit b2,
 ↪   O_Bit b1, O_Bit b0);
```

```
 * \brief Fonction de création d'un octet de 8 bit
 *
 * \param b7: le bit le plus à gauche ou 1er bit
 * \param b6: 2ème bit
 * \param b5: 3ème bit
 * \param b4: 4ème bit
 * \param b3: 5ème bit
 * \param b2: 6ème bit
 * \param b1: 7ème bit
 * \param b0: 8ème bit le bit le plus à droite

 * \return O_Octet
 */
O_Octet O_creerOctet(O_Bit b7, O_Bit b6, O_Bit b5, O_Bit b4, O_Bit b3, O_Bit b2, O_Bit
↪  b1, O_Bit b0);

/**
 * \fn O_Bit O_obtenirIemeBit(O_Octet o, unsigned short i);
 * \brief Fonction qui retourne la valeur du bit situé à la ième position
 *
 * \param o : un octet de 8 bits
 * \param i : position du bit dont on cherche la valeur
 * \return O_Octet
 */
O_Bit O_obtenirIemeBit(O_Octet o, unsigned short i);

/**
 * \fn CB_CodeBinaire CB_creerCodeBinaire(O_Bit b)
 * \brief Fonction qui retourne le naturel associé à l'octet o
 *
 * \param o : un octet de 8 bits
 * \return Unsigned char
 */
unsigned char O_octetVersNaturel(O_Octet o);

/**
 * \fn O_Octet O_naturelVersOctet(unsigned char n);
 * \brief Fonction qui retourne l'octet associé à un naturel compris entre 0 et 255
 *
 * \param n : naturel compris entre 0-255
 * \return O_Octet
 */
O_Octet O_naturelVersOctet(unsigned char n);

#endif
```

5.1.2 TAD Statistiques

```
/**
 * \file statistiques.h
 * \brief Implémentation du TAD Statistiques pour le compresseur d'Huffman
```

```
 * \author T. Bauer
 * \date 17/12/2023
 *
 */

#ifndef __STATISTIQUES__
#define __STATISTIQUES__

#include "octet.h"

/**
 * \brief Le type S_Statistiques permet de stocker le nombre d'occurences des 256 octets
 ↪  possiblement présents dans un fichier
 *
 */
typedef unsigned long S_Statistiques[MAX_OCTET];

/**
 * \fn void S_statistiques(S_Statistiques *p_s)
 * \brief Procédure de création de statistiques à occurences nulles
 *
 * \param p_s : un pointeur sur les Statistiques à retourner
 */
void S_statistiques(S_Statistiques *p_s);

/**
 * \fn void S_incrementerOccurence(S_Statistiques *p_s, O_Octet o)
 * \brief Procédure d'incrémentation de l'occurence d'un octet
 *
 * \param p_s : un pointeur sur les Statstiques à modifier
 * \param o : l'octet dont l'occurence sera incrémentée
 */
void S_incrementerOccurence(S_Statistiques *p_s, O_Octet o);

/**
 * \fn void S_fixerOccurence(S_Statistiques *p_s, O_Octet o, unsigned long n)
 * \brief Procédure permettant de fixer le nombre d'occurences d'un octet
 *
 * \param p_s : un pointeur sur les Statstiques à modifier
 * \param o : l'octet dont l'occurence sera fixée
 * \param n : le nombre d'occurences de l'octet
 */
void S_fixerOccurence(S_Statistiques *p_s, O_Octet o, unsigned long n);

/**
 * \fn unsigned long S_obtenirOccurence(S_Statistiques s, O_Octet o)
 * \brief Fonction permettant d'obtenir le nombre d'occurences d'un octet
 *
 * \param s : les statistiques
 * \param o : l'octet
 * \return unsigned long
```

```
 */
unsigned long S_obtenirOccurence(S_Statistiques s, O_Octet o);

#endif
```

5.1.3 TAD FileDePriorite

```
/**
 * \file fileDePrioriteDArbreDeHuffman.h
 * \brief Implémentation du TAD FileDePriorite pour le compresseur d'Huffman
 * \author M. Saunier
 * \date 31/12/2023
 *
 */

#ifndef ___FILE_DE_PRIORITE_D_ARBRE_DE_HUFFMAN__
#define ___FILE_DE_PRIORITE_D_ARBRE_DE_HUFFMAN__

#include <stdbool.h>

#include "arbreDeHuffman.h"

/**
 * \brief Le type FDPAH_FileDePriorite est un pointeur vers un FDPAH_Noeud
 *
 */
typedef struct FDPAH_Noeud *FDPAH_FileDePriorite;

/**
 * \brief Le type FDPAH_Noeud est une structure qui contient 2 champs : un ArbreDeHuffan et
 ↪  une autre FileDePriorite
 *
 */
typedef struct FDPAH_Noeud {
    ADH_ArbreDeHuffman arbre;          /**< l'ArbreDeHuffman contenu dans le noeud */
    FDPAH_FileDePriorite fileSuivante; /**< le pointeur vers le noeud suivant */
} FDPAH_Noeud;

/**
 * \fn FDPAH_FileDePriorite FDPAH_fileDePriorite(void)
 * \brief Fonction créant une FileDePriorite, pointant sur NULL, pour des ArbreDeHuffman
 *
 * \return FDPAH_FileDePriorite
 */
FDPAH_FileDePriorite FDPAH_fileDePriorite(void);

/**
 * \fn bool FDPAH_estVide(FDPAH_FileDePriorite fdp)
 * \brief Fonction renvoyant VRAI si une FileDePriorite est vide, càd qu'elle ne contient aucun
 ↪  ArbreDeHuffman. Retourne FAUX sinon.
 *
```

```
 * \param fdp : FDPAH_FileDePriorite
 * \return Booleen
 */
bool FDPAH_estVide(FDPAH_FileDePriorite fdp);

/**
 * \fn void FDPAH_enfiler(FDPAH_FileDePriorite *p_fdp, ADH_ArbreDeHuffman a)
 * \brief Fonction permettant d'insérer à l'endroit correct (par rapport à l'élément contenu dans la
 ↪   racine de l'arbre) un ArbreDeHuffman dans une FileDePriorite.
 *
 * \param p_fdp : la FileDePriorite à modifier
 * \param a : l'ADH_ArbreDeHuffman à insérer
 */
void FDPAH_enfiler(FDPAH_FileDePriorite *p_fdp, ADH_ArbreDeHuffman a);

/**
 * \fn ADH_ArbreDeHuffman FDPAH_obtenirElementEtDefiler(FDPAH_FileDePriorite *p_fdp)
 * \brief Fonction permettant d'extraire l'ArbreDeHuffman au bout de la FileDePriorite
 *
 * \param p_fdp : FileDePriorite dont on extrait l'ArbreDeHuffman
 * \return ADH_ArbreDeHuffman
 */
ADH_ArbreDeHuffman FDPAH_obtenirElementEtDefiler(FDPAH_FileDePriorite *p_fdp);

#endif
```

5.1.4 TAD ArbreDeHuffman

```
/**
 * \file arbreDeHuffman.h
 * \brief Implémentation du TAD ArbreDeHuffman pour le compresseur d'Huffman
 * \author M. Saunier
 * \date 31/12/2023
 *
 */

#ifndef __ARBRE_DE_HUFFMAN__
#define __ARBRE_DE_HUFFMAN__

#include <stdbool.h>

#include "octet.h"

/**
 * \brief Le type ADH_ArbreDeHuffman est un pointeur vers un ADH_Noeud
 *
 */
typedef struct ADH_Noeud *ADH_ArbreDeHuffman;

/**
 * \brief Le type ADH_Noeud est une structure qui contient un octet ainsi qu'une fréquence
```

```
 * associée à cet octet. La valeur de l'octet n'a de sens que si le noeud est une feuille et cette
 ↪  information
 * est stocée dans le booléen 'estUneFeuille'. Les deux derniers champs de cette structure sont 2
 ↪  ArbreDeHuffman
 * qui représentent un fils gauche et un fils droit
 *
 */
typedef struct ADH_Noeud {
    O_Octet octet;                  /**< l'octet contenu dans l'arbre feuille */
    unsigned long frequence;        /**< la fréquence de l'octet si l'arbre est une feuille, la somme
    ↪  des fréquences de ses fils sinon */
    bool estUneFeuille;             /**< booléen permettant de savoir si l'arbre est une feuille */
    ADH_ArbreDeHuffman arbreGauche; /**< le sous-arbre gauche */
    ADH_ArbreDeHuffman arbreDroit;  /**< le sous-arbre droit */
} ADH_Noeud;

/**
 * \fn ADH_ArbreDeHuffman ADH_arbreDeHuffman(O_Octet o, unsigned long n)
 * \brief Fonction de création d'un ArbreDeHuffman "feuille" (avec un octet et une fréquence
 ↪  associée)
 *
 * \param o : l'octet
 * \param n : la fréquence associé à l'octet
 * \return ADH_ArbreDeHuffman
 */
ADH_ArbreDeHuffman ADH_arbreDeHuffman(O_Octet o, unsigned long n);

/**
 * \fn ADH_ArbreDeHuffman ADH_fusionner(ADH_ArbreDeHuffman ag, ADH_ArbreDeHuffman
 ↪  ad)
 * \brief Fonction permettant la fusion de 2 ArbreDeHuffman avec une racine qui a pour fréquence
 ↪  la somme des fréquences des 2 ArbreDeHuffman
 *
 * \param ag : ADH_ArbreDeHuffman
 * \param ad : ADH_ArbreDeHuffman
 * \return ADH_ArbreDeHuffman
 */
ADH_ArbreDeHuffman ADH_fusionner(ADH_ArbreDeHuffman ag, ADH_ArbreDeHuffman ad);

/**
 * \fn ADH_estUneFeuille(ADH_ArbreDeHuffman a)
 * \brief Fonction permettant de savoir si un ArbreDeHuffman est une feuille
 *
 * \param a : ADH_ArbreDeHuffman
 * \return Booleen
 */
bool ADH_estUneFeuille(ADH_ArbreDeHuffman a);

/**
 * \fn ADH_obtenirOctet(ADH_ArbreDeHuffman a)
 * \brief Fonction permettant d'obtenir l'octet d'un ArbreDeHuffman
```

```
 *
 * \param a : ADH_ArbreDeHuffman
 * \return O_Octet
 */
O_Octet ADH_obtenirOctet(ADH_ArbreDeHuffman a);

/**
 * \fn ADH_obtenirFrequence(ADH_ArbreDeHuffman a)
 * \brief Fonction permettant d'obtenir la fréquence d'un ArbreDeHuffman
 *
 * \param a : ADH_ArbreDeHuffman
 * \return unsigned long
 */
unsigned long ADH_obtenirFrequence(ADH_ArbreDeHuffman a);

/**
 * \fn ADH_obtenirFilsGauche(ADH_ArbreDeHuffman a)
 * \brief Fonction permettant d'obtenir le fils gauche d'un ArbreDeHuffman
 *
 * \param a : ADH_ArbreDeHuffman
 * \return ADH_ArbreDeHuffman
 */
ADH_ArbreDeHuffman ADH_obtenirFilsGauche(ADH_ArbreDeHuffman a);

/**
 * \fn ADH_obtenirFilsDroit(ADH_ArbreDeHuffman a)
 * \brief Fonction permettant d'obtenir le fils droit d'un ArbreDeHuffman
 *
 * \param a : ADH_ArbreDeHuffman
 * \return ADH_ArbreDeHuffman
 */
ADH_ArbreDeHuffman ADH_obtenirFilsDroit(ADH_ArbreDeHuffman a);

/**
 * \fn ADH_liberer(ADH_ArbreDeHuffman a)
 * \brief Fonction permettant de libérer la mémoire associée à un ArbreDeHuffman
 *
 * \param a : ADH_ArbreDeHuffman
 */
void ADH_liberer(ADH_ArbreDeHuffman a);

#endif
```

5.1.5 TAD CodeBinaire

```
/**
 * \file codeBinaire.h
 * \brief Implémentation du TAD CodeBinaire pour le compresseur d'Huffman
 * \author T. Bauer
 * \date 17/12/2023
 *
```

```
 */

#ifndef __CODE_BINAIRE__
#define __CODE_BINAIRE__

#include "octet.h"

#define MAX_CB (8 * sizeof(unsigned long long))

/**
 * \brief Le type CB_CodeBinaire permet de stocker des bits à la suite
 *
 */
typedef struct CB_CodeBinaire {
    unsigned long long codeBinaire; /**< les octets (stockés sous forme de naturel) contenant ces
    ↪  bits */
    unsigned short nbBits;          /**< le nombre de bits du code binaire */
} CB_CodeBinaire;

/**
 * \fn CB_CodeBinaire CB_creerCodeBinaire(O_Bit b)
 * \brief Fonction de création d'un code binaire à 1 bit
 *
 * \param b : le bit
 * \return CB_CodeBinaire
 */
CB_CodeBinaire CB_creerCodeBinaire(O_Bit b);

/**
 * \fn void CB_ajouterBit(CB_CodeBinaire *p_cb, O_Bit b)
 * \brief Procédure permettant d'ajouter un bit à un code binaire
 *
 * \param p_cb : un pointeur sur le code binaire à modifier
 * \param b : le bit à ajouter
 */
void CB_ajouterBit(CB_CodeBinaire *p_cb, O_Bit b);

/**
 * \fn O_Bit CB_obtenirIemeBit(CB_CodeBinaire cb, unsigned short i)
 * \brief Fonction permettant de retourner le bit à d'indice i d'un code binaire
 *
 * \param cb : le code binaire
 * \param i : l'indice du bit à retourner
 * \return CB_CodeBinaire
 */
O_Bit CB_obtenirIemeBit(CB_CodeBinaire cb, unsigned short i);

/**
 * \fn unsigned short CB_obtenirLongueur(CB_CodeBinaire cb)
 * \brief Fonction permettant d'obtenir le nombre de bits présents dans un code binaire
 *
```

```
 * \param cb : le code binaire
 * \return unsigned short
 */
unsigned short CB_obtenirLongueur(CB_CodeBinaire cb);

#endif
```

5.1.6 TAD TableDeCodage

```
/**
 * \file statistiques.h
 * \brief Implémentation du TAD TableDeCodage pour le compresseur d'Huffman
 * \author O.Taoba
 * \date 07/01/2024
 *
 */
#ifndef __TABLE_DE_CODAGE__
#define __TABLE_DE_CODAGE__

#include <stdbool.h>

#include "codeBinaire.h"
#include "octet.h"
/**
 * \brief Le type TDC_TableDeCodage est une structure contenant 2 champs : un tableau de
 ↪  CodeBinaire et un Tableau de Booleen qui donne la présence d'un code binaire associé à un
 ↪  octet particulier
 *
 */
typedef struct TDC_TableDeCodage {
    CB_CodeBinaire tableDeCodeBinaire[MAX_OCTET];   /**< le tableau contenant les codes
    ↪  binaires */
    bool tableDePresence[MAX_OCTET];                /**< le tableau permettant de connaître la
    ↪  présence ou non des codes binaires */
} TDC_TableDeCodage;

/**
 * \fn TDC_TableDeCodage TDC_creerTableCodage(void)
 * \brief Fonction permettant de créer une TableDeCodage Vide
 *
 * \return TDC_TableDeCodage
 */
TDC_TableDeCodage TDC_creerTableCodage();

/**
 * \fn void TDC_ajouterCodage(TDC_TableDeCodage* p_tdc, O_Octet o, CB_CodeBinaire cb)
 * \brief Fonction permettant d'ajouter un codeBinaire à une table de codage
 *
 * \param p_tdc : un pointeur sur la TableDeCodage à modifier
 * \param o : l'Octet correspondant au CodeBinaire à ajouter
 * \param cb : le CodeBinaire à ajouter à la TableDeCodage
```

```
 */
void TDC_ajouterCodage(TDC_TableDeCodage* p_tdc, O_Octet o, CB_CodeBinaire cb);
/**
 * \fn CB_CodeBinaire TDC_octetVersCodeBinaire(TDC_TableDeCodage tdc, O_Octet o)
 * \brief Procédure permettant de retourner le CodeBinaire associé à un Octet en entrée à partir
 ↪  d'une TableDeCodage
 *
 * \param tdc : la table de codage permettant de récupérer les données
 * \param o : l'octet dont on souhaite recupérer le code binaire
 * \return CB_CodeBinaire
 */
CB_CodeBinaire TDC_octetVersCodeBinaire(TDC_TableDeCodage tdc, O_Octet o);
/**
 * \fn bool TDC_octetPresent(TDC_TableDeCodage tdc, O_Octet o)
 * \brief Fonction permettant de savoir si le CodeBinaire d'un Octet est présent dans une table de
 ↪  codage
 *
 * \param tdc : la table de codage permettant de récupérer les données
 * \param o : l'Octet dont on veux vérifier la présence
 * \return Booleen
 */
bool TDC_octetPresent(TDC_TableDeCodage tdc, O_Octet o);

#endif
```

5.1.7 Construction de l'arbre de Huffman

```
/**
 * \file construireArbreDeHuffman.h
 * \brief Implémentation des fonctions utilisées pour construire un arbre de Huffman à partir de
 ↪  statistiques
 * \author T. Bauer
 * \date 19/12/2023
 *
 */

#ifndef ___CONSTRUIRE_ARBRE_DE_HUFFMAN___
#define ___CONSTRUIRE_ARBRE_DE_HUFFMAN___

#include "arbreDeHuffman.h"
#include "fileDePrioriteDArbreDeHuffman.h"
#include "statistiques.h"

/**
 * \fn FDPAH_FileDePriorite CADH_construireFileDePriorite(S_Statistiques s)
 * \brief Fonction permettant de construire la file de priorité contenant les feuilles d'occurences non
 ↪  nulles à partir de statistiques
 *
 * \param s : les statistiques
 * \return FDPAH_FileDePriorite
 */
```

```
FDPAH_FileDePriorite CADH_construireFileDePriorite(S_Statistiques s);

/**
 * \fn ADH_ArbreDeHuffman CADH_construireArbreDeHuffman(S_Statistiques s)
 * \brief Fonction permettant de construire l'arbre de Huffman à partir de statistiques
 *
 * \param s : les statistiques
 * \return ADH_ArbreDeHuffman
 */
ADH_ArbreDeHuffman CADH_construireArbreDeHuffman(S_Statistiques s);

#endif
```

5.1.8 Compression

```
/**
 * \file compression.h
 * \brief Implémentation de la fonction compresser pour le compresseur d'Huffman
 * \author O.Taoba
 * \date 07/01/2024
 *
 */

#ifndef __COMPRESSION__
#define __COMPRESSION__

#include <stddef.h>
#include <stdio.h>

#define IDENTIFIANT 1000

/**
 * \fn C_Compresser(FILE *f, char *filename)
 * \brief Fonction permettant la compression d'un fichier par la méthode du codage de Huffman
 *
 * \param f : le fichier à compresser
 * \param filename : le nom du fichier à compresser
 */
void C_compresser(FILE *f, char *filename);

#endif
```

5.1.9 Décompression

```
/**
 * \file decompression.h
 * \brief Implémentation de la fonction decompresser pour le compresseur d'Huffman
 * \author M. Saunier
 * \date 31/12/2023
 *
 */
```

```c
#ifndef __DECOMPRESSION__
#define __DECOMPRESSION__

#include <stdio.h>

/**
 * \fn D_decompresser(FILE *fbCompresse, char *filename)
 * \brief Fonction permettant la décompression d'un fichier, dont le nom est donné en entrée,
 ↪  préalablement compressé par notre algortithme du compresseur d'Huffman. Le fichier
 ↪  décompressé est crée par la fonction et portera le nom du fichier compressé sans l'extension
 ↪  '.huff'
 *
 * \param fbCompresse : le FichierBinaire compressé
 * \param filename : le nom du fichier compressé
 */
void D_decompresser(FILE *fbCompresse, char *filename);

#endif
```

5.2 Code source

5.2.1 TAD Octet

```c
#include "octet.h"

#include <assert.h>
#include <stdio.h>

O_Octet O_creerOctet(O_Bit b7, O_Bit b6, O_Bit b5, O_Bit b4, O_Bit b3, O_Bit b2, O_Bit
↪  b1, O_Bit b0) {
    return b0 + (b1 << 1) + (b2 << 2) + (b3 << 3) + (b4 << 4) + (b5 << 5) + (b6 << 6)
    ↪  + (b7 << 7);
}

O_Bit O_obtenirIemeBit(O_Octet o, unsigned short i) {
    assert(i < MAX_BITS);
    return (o >> i) & 1;
    // L'opérateur >> est un décalage de i vers la droite de l'octet o, ce qui correspond à une
    ↪  divison par 2^i
    // L'opérateur & compare bit à bit : en comparant avec 1 (0000001), on obtient la parité de
    ↪  l'opérande de gauche, ce qui correspond à un modulo 2
}

unsigned char O_octetVersNaturel(O_Octet o) {
    return o;
}

O_Octet O_naturelVersOctet(unsigned char n) {
    return n;
}
```

5.2.2 TAD Statistiques

```c
#include "statistiques.h"

void S_statistiques(S_Statistiques *p_s) {
    for (unsigned short o = 0; o < MAX_OCTET; o++)
        (*p_s)[o] = 0;
}

void S_incrementerOccurence(S_Statistiques *p_s, O_Octet o) {
    (*p_s)[O_octetVersNaturel(o)]++;
}

void S_fixerOccurence(S_Statistiques *p_s, O_Octet o, unsigned long n) {
    (*p_s)[O_octetVersNaturel(o)] = n;
}

unsigned long S_obtenirOccurence(S_Statistiques s, O_Octet o) {
    return s[O_octetVersNaturel(o)];
```

```
}
```

5.2.3 TAD FileDePriorite

```
#include "fileDePrioriteDArbreDeHuffman.h"

#include <assert.h>
#include <stddef.h>
#include <stdlib.h>

#include "arbreDeHuffman.h"

FDPAH_FileDePriorite FDPAH_fileDePriorite() {
    return NULL;
}

bool FDPAH_estVide(FDPAH_FileDePriorite fdp) {
    return (fdp == NULL);
}

void FDPAH_enfiler(FDPAH_FileDePriorite *p_fdp, ADH_ArbreDeHuffman a) {
    if (FDPAH_estVide(*p_fdp) || ADH_obtenirFrequence(a) <
    ↪  ADH_obtenirFrequence((*p_fdp)->arbre)
        // Si les deux fréquences sont égales, on compare (si on peut) leurs octets
        || (ADH_obtenirFrequence(a) == ADH_obtenirFrequence((*p_fdp)->arbre)
            && ADH_estUneFeuille(a) && ADH_estUneFeuille((*p_fdp)->arbre)
            && O_octetVersNaturel(ADH_obtenirOctet(a)) <
            ↪  O_octetVersNaturel(ADH_obtenirOctet((*p_fdp)->arbre))
            )
        ) {
        FDPAH_FileDePriorite temp = (FDPAH_FileDePriorite)malloc(sizeof(FDPAH_Noeud));
        temp->arbre = a;
        temp->fileSuivante = *p_fdp;
        *p_fdp = temp;
    } else {
        FDPAH_enfiler(&((*p_fdp)->fileSuivante), a);
    }
}

ADH_ArbreDeHuffman FDPAH_obtenirElementEtDefiler(FDPAH_FileDePriorite *fdp) {
    assert(!FDPAH_estVide(*fdp));
    ADH_ArbreDeHuffman a = (*fdp)->arbre;
    FDPAH_FileDePriorite temp = *fdp;
    *fdp = temp->fileSuivante;
    free(temp);
    return a;
}
```

5.2.4 TAD ArbreDeHuffman

```
#include "arbreDeHuffman.h"

#include <assert.h>
#include <stddef.h>
#include <stdlib.h>

ADH_ArbreDeHuffman ADH_arbreDeHuffman(O_Octet o, unsigned long n) {
    ADH_ArbreDeHuffman a = (ADH_ArbreDeHuffman)malloc(sizeof(ADH_Noeud));
    a->octet = o;
    a->frequence = n;
    a->estUneFeuille = true;
    a->arbreGauche = NULL;
    a->arbreDroit = NULL;
    return a;
}

unsigned long ADH_obtenirFrequence(ADH_ArbreDeHuffman a) {
    return a->frequence;
}

ADH_ArbreDeHuffman ADH_fusionner(ADH_ArbreDeHuffman ag, ADH_ArbreDeHuffman ad) {
    ADH_ArbreDeHuffman racine = (ADH_ArbreDeHuffman)malloc(sizeof(ADH_Noeud));
    racine->arbreGauche = ag;
    racine->arbreDroit = ad;
    racine->estUneFeuille = false;
    racine->frequence = ADH_obtenirFrequence(ag) + ADH_obtenirFrequence(ad);
    return racine;
}

bool ADH_estUneFeuille(ADH_ArbreDeHuffman a) {
    return a->estUneFeuille;
}

ADH_ArbreDeHuffman ADH_obtenirFilsGauche(ADH_ArbreDeHuffman a) {
    assert(!ADH_estUneFeuille(a));
    return a->arbreGauche;
}

O_Octet ADH_obtenirOctet(ADH_ArbreDeHuffman a) {
    assert(ADH_estUneFeuille(a));
    return a->octet;
}

ADH_ArbreDeHuffman ADH_obtenirFilsDroit(ADH_ArbreDeHuffman a) {
    assert(!ADH_estUneFeuille(a));
    return a->arbreDroit;
}

void ADH_liberer(ADH_ArbreDeHuffman a) {
    if (!(ADH_estUneFeuille(a))) {
```

```
        ADH_liberer(ADH_obtenirFilsGauche(a));
        ADH_liberer(ADH_obtenirFilsDroit(a));
    }
    free(a);
}
```

5.2.5 TAD CodeBinaire

```
#include "codeBinaire.h"

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

CB_CodeBinaire CB_creerCodeBinaire(O_Bit b) {
    CB_CodeBinaire cb;
    cb.codeBinaire = b;
    cb.nbBits = 1;
    return cb;
}

unsigned short CB_obtenirLongueur(CB_CodeBinaire cb) {
    return cb.nbBits;
}

void CB_ajouterBit(CB_CodeBinaire *cb, O_Bit b) {
    cb->codeBinaire = cb->codeBinaire + (b << cb->nbBits);
    cb->nbBits++;
}

O_Bit CB_obtenirIemeBit(CB_CodeBinaire cb, unsigned short i) {
    assert(i < cb.nbBits);
    return (cb.codeBinaire >> i) & 1;
}
```

5.2.6 TAD TableDeCodage

```
#include "tableDeCodage.h"

#include <assert.h>

#include "octet.h"

TDC_TableDeCodage TDC_creerTableCodage() {
    TDC_TableDeCodage tdc;
    for (unsigned short octet = 0; octet < MAX_OCTET; octet++) {
        tdc.tableDePresence[octet] = false;
    }
    return tdc;
}
```

```
bool TDC_octetPresent(TDC_TableDeCodage tdc, O_Octet o) {
    return tdc.tableDePresence[O_octetVersNaturel(o)];
}

void TDC_ajouterCodage(TDC_TableDeCodage *tdc, O_Octet o, CB_CodeBinaire cb) {
    assert(!TDC_octetPresent(*tdc, o));
    unsigned short octet = O_octetVersNaturel(o);
    tdc->tableDeCodeBinaire[octet] = cb;
    tdc->tableDePresence[octet] = true;
}

CB_CodeBinaire TDC_octetVersCodeBinaire(TDC_TableDeCodage tdc, O_Octet o) {
    return tdc.tableDeCodeBinaire[O_octetVersNaturel(o)];
}
```

5.2.7 Fonctions communes à la compression et à la décompression

```
#include <stdio.h>

#include "arbreDeHuffman.h"
#include "fileDePrioriteDArbreDeHuffman.h"
#include "octet.h"
#include "statistiques.h"

FDPAH_FileDePriorite CADH_construireFileDePriorite(S_Statistiques s) {
    FDPAH_FileDePriorite fdp;
    O_Octet octet;
    unsigned long occurence;

    fdp = FDPAH_fileDePriorite();

    unsigned short o;
    for (o = 0; o < MAX_OCTET; o++) {
        octet = O_naturelVersOctet(o);
        occurence = S_obtenirOccurence(s, octet);

        if (occurence > 0) {
            FDPAH_enfiler(&fdp, ADH_arbreDeHuffman(octet, occurence));
        }
    }

    return fdp;
}

ADH_ArbreDeHuffman CADH_construireArbreDeHuffman(S_Statistiques s) {
    FDPAH_FileDePriorite fdp;
    bool dernierElement;
    ADH_ArbreDeHuffman a1, a2, aFusion;

    fdp = CADH_construireFileDePriorite(s);
    dernierElement = false;
```

```
    while (!(dernierElement)) {
        a1 = FDPAH_obtenirElementEtDefiler(&fdp);
        if (FDPAH_estVide(fdp)) {
            dernierElement = true;
        } else {
            a2 = FDPAH_obtenirElementEtDefiler(&fdp);
            aFusion = ADH_fusionner(a1, a2);
            FDPAH_enfiler(&fdp, aFusion);
        }
    }
    return a1;
}
```

5.2.8 Compression

```
#include "compression.h"

#include <assert.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>

#include "arbreDeHuffman.h"
#include "codeBinaire.h"
#include "construireArbreDeHuffman.h"
#include "fileDePrioriteDArbreDeHuffman.h"
#include "octet.h"
#include "statistiques.h"
#include "tableDeCodage.h"

unsigned short min(unsigned short a, unsigned short b) {
    if (a > b)
        return b;
    else
        return a;
}

void C_obtenirStatistiquesEtTailleFichier(FILE *f, S_Statistiques *s, unsigned long long *taille)
↪  {
    rewind(f);

    S_statistiques(s);
    *taille = 0;

    short o;
    while ((o = fgetc(f)) != EOF) {
        S_incrementerOccurence(s, O_naturelVersOctet(o));
        (*taille)++;
    }
}
```

```
void C_obtenirTableDeCodageRecursif(TDC_TableDeCodage *tdc, ADH_ArbreDeHuffman a,
↪  CB_CodeBinaire cb) {
    CB_CodeBinaire cbCopie;

    if (ADH_estUneFeuille(a)) {
        TDC_ajouterCodage(tdc, ADH_obtenirOctet(a), cb);
    } else {
        memcpy(&cbCopie, &cb, sizeof(CB_CodeBinaire));

        CB_ajouterBit(&cbCopie, bitA0);
        C_obtenirTableDeCodageRecursif(tdc, ADH_obtenirFilsGauche(a), cbCopie);
        CB_ajouterBit(&cb, bitA1);
        C_obtenirTableDeCodageRecursif(tdc, ADH_obtenirFilsDroit(a), cb);
    }
}

TDC_TableDeCodage C_obtenirTableDeCodage(ADH_ArbreDeHuffman a) {
    assert(!ADH_estUneFeuille(a));

    TDC_TableDeCodage tdc = TDC_creerTableCodage();

    CB_CodeBinaire cbGauche = CB_creerCodeBinaire(bitA0);
    CB_CodeBinaire cbDroit = CB_creerCodeBinaire(bitA1);
    C_obtenirTableDeCodageRecursif(&tdc, ADH_obtenirFilsGauche(a), cbGauche);
    C_obtenirTableDeCodageRecursif(&tdc, ADH_obtenirFilsDroit(a), cbDroit);

    return tdc;
}

void C_ecrireIdentifiant(FILE *f) {
    unsigned short identifiant = IDENTIFIANT;
    fwrite(&identifiant, sizeof(unsigned short), 1, f);
}

void C_ecrireTailleFichier(FILE *f, unsigned long long taille) {
    fwrite(&taille, sizeof(unsigned long long), 1, f);
}

void C_ecrireStatistiques(FILE *f, S_Statistiques s) {
    O_Octet octet;
    unsigned long occurence;

    unsigned short o;
    for (o = 0; o < MAX_OCTET; o++) {
        octet = O_naturelVersOctet(o);
        occurence = S_obtenirOccurence(s, octet);
        if (occurence > 0) {
            fwrite(&occurence, sizeof(unsigned long), 1, f);
            fwrite(&o, sizeof(unsigned char), 1, f);
        }
    }
```

```
    occurence = 0;
    fwrite(&occurence, sizeof(unsigned long), 1, f);
}

O_Octet C_codeBinaireEnOctet(CB_CodeBinaire cb) {
    assert(CB_obtenirLongueur(cb) == MAX_BITS);
    return O_creerOctet(CB_obtenirIemeBit(cb, 7),
                        CB_obtenirIemeBit(cb, 6),
                        CB_obtenirIemeBit(cb, 5),
                        CB_obtenirIemeBit(cb, 4),
                        CB_obtenirIemeBit(cb, 3),
                        CB_obtenirIemeBit(cb, 2),
                        CB_obtenirIemeBit(cb, 1),
                        CB_obtenirIemeBit(cb, 0));
}

void C_concatenerCodeBinaireDansFichier(FILE *f, CB_CodeBinaire *p_cbTemp,
↪   CB_CodeBinaire cb) {
    unsigned short i = 0;
    unsigned short tailleCb = CB_obtenirLongueur(cb);

    while (i != tailleCb) {
        if (CB_obtenirLongueur(*p_cbTemp) == MAX_BITS) {
            *p_cbTemp = CB_creerCodeBinaire(CB_obtenirIemeBit(cb, i));
            i++;
        }

        while (i < tailleCb && CB_obtenirLongueur(*p_cbTemp) < MAX_BITS) {
            CB_ajouterBit(p_cbTemp, CB_obtenirIemeBit(cb, i));
            i++;
        }

        if (CB_obtenirLongueur(*p_cbTemp) == MAX_BITS) {
            unsigned char octet = O_octetVersNaturel(C_codeBinaireEnOctet(*p_cbTemp));
            fwrite(&octet, sizeof(unsigned char), 1, f);
        }
    }
}

void C_encoder(FILE *f, FILE *fbCompresse, TDC_TableDeCodage tdc) {
    unsigned short i;
    rewind(f);

    // Création d'un code binaire temporaire initalisé à 8 bits pour rentrer dans la première
    ↪   condition de la fonction concatenerCodeBinaireEnOctet
    CB_CodeBinaire cbTemp = CB_creerCodeBinaire(bitA0);
    for (i = 1; i < MAX_BITS; i++)
        CB_ajouterBit(&cbTemp, bitA0);

    // Boucle d'encodage
    CB_CodeBinaire cb;
```

```
    short o;
    while ((o = fgetc(f)) != EOF) {
        cb = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet(o));
        C_concatenerCodeBinaireDansFichier(fbCompresse, &cbTemp, cb);
    }

    // Ecriture du dernier octet si le dernier code binaire n'est pas de taille 8 et n'a donc pas été
    ↪  écrit dans le fichier compressé
    if (CB_obtenirLongueur(cbTemp) < MAX_BITS) {
        for (i = CB_obtenirLongueur(cbTemp); i < MAX_BITS; i++)
            CB_ajouterBit(&cbTemp, bitA0);
        unsigned char octet = O_octetVersNaturel(C_codeBinaireEnOctet(cbTemp));
        fwrite(&octet, sizeof(unsigned char), 1, fbCompresse);
    }
}

void C_compresser(FILE *f, char *filename) {
    S_Statistiques s;
    unsigned long long taille;

    rewind(f);
    FILE *fbCompresse = fopen(strcat(filename, ".huff"), "wb");

    C_obtenirStatistiquesEtTailleFichier(f, &s, &taille);

    // Ecriture des données importantes avant d'encoder
    C_ecrireIdentifiant(fbCompresse);
    C_ecrireTailleFichier(fbCompresse, taille);
    if (taille > 0) {  // Cas particulier d'un fichier vide
        C_ecrireStatistiques(fbCompresse, s);
        ADH_ArbreDeHuffman a = CADH_construireArbreDeHuffman(s);
        if (!ADH_estUneFeuille(a))  // Cas particulier d'un fichier contenant un seul octet (présent
        ↪  plusieurs fois ou non)
            C_encoder(f, fbCompresse, C_obtenirTableDeCodage(a));
        ADH_liberer(a);
    }

    fclose(fbCompresse);
}
```

5.2.9 Décompression

```
#include "decompression.h"

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include "arbreDeHuffman.h"
```

```c
#include "construireArbreDeHuffman.h"
#include "fileDePrioriteDArbreDeHuffman.h"
#include "octet.h"
#include "statistiques.h"
#include "compression.h" // On inclut compression.h pour avoir la constante de l'identifiant

void D_seDeplacerDansLArbre(O_Bit b, ADH_ArbreDeHuffman *a) {
    if (b == bitA0) {
        *a = ADH_obtenirFilsGauche(*a);
    } else {
        *a = ADH_obtenirFilsDroit(*a);
    }
}

void D_lireStatistiques(FILE *fb, S_Statistiques *s) {
    unsigned char octet;
    unsigned long occurence;

    S_statistiques(s);
    do {
        // Note : la valeur de type size_t retournée par la fonction fread est le nombre de blocs lus.
        ↪  Si elle est inférieure au nombre de blocs
        // à lire indiqué en paramètre, c'est que nous sommes arrivés à la fin du fichier ou qu'une
        ↪  erreur est survenue.
        size_t nbBlocsLus = fread(&occurence, sizeof(unsigned long int), 1, fb);
        if (nbBlocsLus < 1) {
            printf("Erreur : problème de lecture. Cela peut être causé par un fichier corrumpu.\n");
            exit(EXIT_FAILURE);
        }
        if (occurence != 0) {
            size_t nbBlocsLus = fread(&octet, sizeof(unsigned char), 1, fb);
            if (nbBlocsLus < 1) {
                printf("Erreur : problème de lecture. Cela peut être causé par un fichier corrumpu.\n");
                exit(EXIT_FAILURE);
            }
            S_fixerOccurence(s, O_naturelVersOctet(octet), occurence);
        }

    } while (occurence != 0);
}

void D_decoder(ADH_ArbreDeHuffman aHuff, unsigned long long int longueur, FILE
↪  *fbCompresse, FILE *fbDecompresse) {
    rewind(fbDecompresse);
    ADH_ArbreDeHuffman aTemp = aHuff;
    unsigned long long int compteurOctetsDecodes = 0;
    bool finDecodage = false;
    while (!finDecodage) {
        unsigned char o;
        size_t nbBlocsLus = fread(&o, sizeof(unsigned char), 1, fbCompresse);
        if (nbBlocsLus < 1) {
```

```
            printf("Erreur060 decompression.c : fin du fichier atteinte de manière innatendue ou erreur
            ↪  de la fonction fread \n");
            exit(EXIT_FAILURE);
        }
        for (int i = 0; i < MAX_BITS; i++) {
            if (!finDecodage) {  // if qui permet de régler les bugs sur le dernier octets
                O_Bit b = O_obtenirIemeBit(O_naturelVersOctet(o), i);
                D_seDeplacerDansLArbre(b, &aTemp);
                if (ADH_estUneFeuille(aTemp)) {
                    unsigned char oDecode = O_octetVersNaturel(ADH_obtenirOctet(aTemp));
                    fwrite(&oDecode, sizeof(unsigned char), 1, fbDecompresse);
                    aTemp = aHuff;
                    compteurOctetsDecodes = compteurOctetsDecodes + 1;
                    finDecodage = (compteurOctetsDecodes == longueur);
                }
            }
        }
    }
}

void D_decompresser(FILE *fbCompresse, char *filename) {
    rewind(fbCompresse);
    // Petit bout de code pour supprimer le .huff en fin de la variable filename
    size_t nouvelleLongueurDuNom = strlen(filename) - strlen(".huff");
    filename[nouvelleLongueurDuNom] = '\0';
    FILE *fbDecompresse = fopen(filename, "wb");
    unsigned short int identifiant;
    size_t nbBlocsLus = fread(&identifiant, sizeof(unsigned short int), 1, fbCompresse);
    if (nbBlocsLus < 1) {
        printf("Erreur : problème de lecture. Cela peut être causé par un fichier corrumpu.\n");
        exit(EXIT_FAILURE);
    }
    if (identifiant == IDENTIFIANT) {
        unsigned long long int longueur;
        size_t nbBlocsLus = fread(&longueur, sizeof(unsigned long long int), 1, fbCompresse);
        if (nbBlocsLus < 1) {
            printf("Erreur : problème de lecture. Cela peut être causé par un fichier corrumpu.\n");
            exit(EXIT_FAILURE);
        }
        if (longueur > 0) {  // Cas particulier d'un fichier vide
            S_Statistiques s;
            D_lireStatistiques(fbCompresse, &s);
            ADH_ArbreDeHuffman a = CADH_construireArbreDeHuffman(s);
            if (ADH_estUneFeuille(a)) {  // Cas particulier d'un fichier contenant un seul octet
            ↪  (présent plusieurs fois ou non)
                unsigned char octet = O_octetVersNaturel(ADH_obtenirOctet(a));
                for (unsigned long long i = 1; i <= longueur; i++)
                    fwrite(&octet, sizeof(unsigned char), 1, fbDecompresse);
            } else {
                D_decoder(a, longueur, fbCompresse, fbDecompresse);
            }
```

```
            ADH_liberer(a);
        }
    } else {
        printf("Erreur : identifiant de compression incorrect. Il semble que le fichier ait été compressé
        ↪  à l'origine avec un compresseur différent.");
        exit(EXIT_FAILURE);
    }

    fclose(fbDecompresse);
    chmod(filename, S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
}
```

5.2.10 Programme principal

```
#include <stdlib.h>
#include <string.h>

#include "compression.h"
#include "decompression.h"

void printUtilisation() {
    printf("Utilisation :\n");
    printf("\t- pour compresser : huffman c nomFichier\n");
    printf("\t- pour décompresser : huffman d nomFichier.huff\n");
}

int main(int argc, char *argv[]) {
    if (argc == 3) {
        if (strcmp(argv[1], "c") == 0) {
            FILE *f = fopen(argv[2], "rb");
            if (f != NULL) {
                C_compresser(f, argv[2]);
                fclose(f);
            } else {
                printf("Erreur : fichier inexistant ou corrompu.\n");
                printUtilisation();
                exit(EXIT_FAILURE);
            }
        } else if (strcmp(argv[1], "d") == 0) {
            if (strstr(argv[2], ".huff") != NULL) {
                FILE *f = fopen(argv[2], "rb");
                if (f != NULL) {
                    D_decompresser(f, argv[2]);
                    fclose(f);
                } else {
                    printf("Erreur : fichier inexistant ou corrompu.\n");
                    printUtilisation();
                    exit(EXIT_FAILURE);
                }
            } else {
                printf("Erreur : impossible de décompresser un fichier non compressé.\n");
```

```
                printUtilisation();
                exit(EXIT_FAILURE);
            }
        } else {
            printf("Erreur : deuxième paramètre incorrect.\n");
            printUtilisation();
            exit(EXIT_FAILURE);
        }
    } else {
        printf("Erreur : nombre de paramètres incorrect.\n");
        printUtilisation();
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

5.3 Tests unitaires

5.3.1 Tests des TADs

```c
#include <CUnit/Basic.h>
#include <stdio.h>
#include <stdlib.h>

#include "arbreDeHuffman.h"
#include "codeBinaire.h"
#include "fileDePrioriteDArbreDeHuffman.h"
#include "octet.h"
#include "statistiques.h"
#include "tableDeCodage.h"

int init_suite_success(void) {
    return 0;
}

int clean_suite_success(void) {
    return 0;
}

/* Tests statistiques.c */

void test_statistiques_vides(void) {
    S_Statistiques s;
    S_statistiques(&s);

    for (unsigned short o = 0; o < MAX_OCTET; o++) {
        CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet(o)), 0);
    }
}

void test_statistiques_incrementees(void) {
    S_Statistiques s;
    S_statistiques(&s);

    O_Octet o = O_naturelVersOctet(241);
    unsigned long ancienneOccurence = S_obtenirOccurence(s, o);

    S_incrementerOccurence(&s, o);
    unsigned long nouvelleOccurence = S_obtenirOccurence(s, o);

    CU_ASSERT_EQUAL(nouvelleOccurence, ancienneOccurence + 1);
}

void test_statistiques_fixer_occurence(void) {
    S_Statistiques s;
    S_statistiques(&s);
```

```
    O_Octet o = O_naturelVersOctet(241);
    unsigned long n = 1234;
    S_fixerOccurence(&s, o, n);

    CU_ASSERT_EQUAL(S_obtenirOccurence(s, o), n);
}

/* Tests codeBinaire.c */

void test_creation_codebinaire(void) {
    O_Bit b = bitA0;
    CB_CodeBinaire cb = CB_creerCodeBinaire(b);
    CU_ASSERT_EQUAL(CB_obtenirLongueur(cb), 1);
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 0), b);
}

void test_ajout_bit(void) {
    CB_CodeBinaire cb = CB_creerCodeBinaire(bitA0);
    unsigned short ancienneLongueur = CB_obtenirLongueur(cb);
    O_Bit b = bitA1;
    CB_ajouterBit(&cb, b);
    unsigned short nouvelleLongueur = CB_obtenirLongueur(cb);
    CU_ASSERT_EQUAL(nouvelleLongueur, ancienneLongueur + 1);
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, nouvelleLongueur - 1), b);
}

/* Tests arbreDeHuffman.c */

void test_creation_arbre_de_huffman_feuille(void) {
    unsigned char o = 65;
    unsigned long f = 2;

    ADH_ArbreDeHuffman a = ADH_arbreDeHuffman(O_naturelVersOctet(o), f);

    CU_ASSERT(ADH_estUneFeuille(a));
    CU_ASSERT_EQUAL(ADH_obtenirFrequence(a), f);
    CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(a)), o);

    ADH_liberer(a);
}

void test_fusionner_ADH(void) {
    O_Octet og = O_naturelVersOctet(241);
    unsigned long ng = 2;
    O_Octet od = O_naturelVersOctet(121);
    unsigned long nd = 3;

    ADH_ArbreDeHuffman ad = ADH_arbreDeHuffman(od, nd);
    ADH_ArbreDeHuffman ag = ADH_arbreDeHuffman(og, ng);
    ADH_ArbreDeHuffman a = ADH_fusionner(ad, ag);
```

```
    CU_ASSERT_EQUAL(ADH_obtenirFrequence(a), ADH_obtenirFrequence(ag) +
    ↪  ADH_obtenirFrequence(ad));
    CU_ASSERT_EQUAL(ADH_obtenirFilsDroit(a), ag);
    CU_ASSERT_EQUAL(ADH_obtenirFilsGauche(a), ad);
    CU_ASSERT(!ADH_estUneFeuille(a));
}

/* Tests FileDePriorite.c */

void test_creation_filedePriorite_vide(void) {
    FDPAH_FileDePriorite fdp = FDPAH_fileDePriorite();

    CU_ASSERT(FDPAH_estVide(fdp));
}

void test_enfiler(void) {
    FDPAH_FileDePriorite fdp = FDPAH_fileDePriorite();

    ADH_ArbreDeHuffman a1 = ADH_arbreDeHuffman(O_naturelVersOctet('A'), 10);

    FDPAH_enfiler(&fdp, a1);

    CU_ASSERT_FALSE(FDPAH_estVide(fdp));

    ADH_liberer(a1);
}

void test_obtenir_element_et_defiler(void) {
    O_Octet o1 = O_naturelVersOctet('G');
    unsigned long f1 = 3;
    O_Octet o2 = O_naturelVersOctet('A');
    unsigned long f2 = 2;
    O_Octet o3 = O_naturelVersOctet('D');
    unsigned long f3 = 3;

    FDPAH_FileDePriorite fdp;
    fdp = FDPAH_fileDePriorite();
    ADH_ArbreDeHuffman a1 = ADH_arbreDeHuffman(o1, f1);
    ADH_ArbreDeHuffman a2 = ADH_arbreDeHuffman(o2, f2);
    ADH_ArbreDeHuffman a3 = ADH_arbreDeHuffman(o3, f3);

    FDPAH_enfiler(&fdp, a1);
    FDPAH_enfiler(&fdp, a2);
    FDPAH_enfiler(&fdp, a3);

    CU_ASSERT_EQUAL(FDPAH_obtenirElementEtDefiler(&fdp), a2);
    CU_ASSERT_EQUAL(FDPAH_obtenirElementEtDefiler(&fdp), a3);
    CU_ASSERT_EQUAL(FDPAH_obtenirElementEtDefiler(&fdp), a1);
    CU_ASSERT(FDPAH_estVide(fdp));

    ADH_liberer(a1);
```

```
    ADH_liberer(a2);
    ADH_liberer(a3);
}

/* Tests tableDeCodage.c */

void test_creerTableCodage(void) {
    /* Utilisation d'un do while ... pour éviter le rique d'un dépassement de mémoire (256 avec un
    ↪  unsigned char) */
    TDC_TableDeCodage tdc = TDC_creerTableCodage();
    unsigned char o = 0;
    do {
        CU_ASSERT(!(TDC_octetPresent(tdc, O_naturelVersOctet(o))));
        o++;
    } while (o != MAX_OCTET - 1);
}

void test_ajouterCodage(void) {
    TDC_TableDeCodage tdc = TDC_creerTableCodage();
    CB_CodeBinaire cb_42_test = CB_creerCodeBinaire(bitA1);
    TDC_ajouterCodage(&tdc, O_naturelVersOctet(42), cb_42_test);
    CB_CodeBinaire cb_43_test = CB_creerCodeBinaire(bitA1);
    CB_ajouterBit(&cb_43_test, bitA0);
    TDC_ajouterCodage(&tdc, O_naturelVersOctet(43), cb_43_test);

    unsigned char i = 0;
    do {
        if (i == 42) {
            CU_ASSERT(TDC_octetPresent(tdc, O_naturelVersOctet(i)));
            CB_CodeBinaire cb_42_lu = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet(i));
            CU_ASSERT((CB_obtenirIemeBit(cb_42_lu, 0) == CB_obtenirIemeBit(cb_42_test,
            ↪  0)));
            CU_ASSERT((CB_obtenirLongueur(cb_42_lu) == CB_obtenirLongueur(cb_42_test)));
            CU_ASSERT((CB_obtenirLongueur(cb_42_lu) == 1));
        } else if (i == 43) {
            CU_ASSERT(TDC_octetPresent(tdc, O_naturelVersOctet(i)));
            CB_CodeBinaire cb_43_lu = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet(i));
            CU_ASSERT((CB_obtenirIemeBit(cb_43_lu, 0) == CB_obtenirIemeBit(cb_43_test,
            ↪  0)));
            CU_ASSERT((CB_obtenirIemeBit(cb_43_lu, 1) == CB_obtenirIemeBit(cb_43_test,
            ↪  1)));
            CU_ASSERT((CB_obtenirLongueur(cb_43_lu) == CB_obtenirLongueur(cb_43_test)));
            CU_ASSERT((CB_obtenirLongueur(cb_43_lu) == 2));
        } else {
            CU_ASSERT(!(TDC_octetPresent(tdc, O_naturelVersOctet(i))));
        }
        i = i + 1;
    } while (i != 255);
}

void test_octetVersCodeBinaire(void) {
```

```
    TDC_TableDeCodage tdc = TDC_creerTableCodage();
    CB_CodeBinaire cb_42_test = CB_creerCodeBinaire(bitA1);
    TDC_ajouterCodage(&tdc, O_naturelVersOctet(42), cb_42_test);
    unsigned char i = 0;
    do {
        if (i == 42) {
            CU_ASSERT(TDC_octetPresent(tdc, O_naturelVersOctet(i)));
            CB_CodeBinaire cb = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet(42));
            CU_ASSERT((CB_obtenirIemeBit(cb, 0) == bitA1));
            CU_ASSERT((CB_obtenirLongueur(cb) == 1));
        } else {
            CU_ASSERT(!(TDC_octetPresent(tdc, O_naturelVersOctet(i))));
        }
        i = i + 1;
    } while (i != 255);
}

/* Tests octet.c*/
void test_creer_octet(void) {
    // Test pour 0 (Binaire : 00000000)
    O_Bit b7 = 0, b6 = 0, b5 = 0, b4 = 0, b3 = 0, b2 = 0, b1 = 0, b0 = 0;
    O_Octet resultat = O_creerOctet(b7, b6, b5, b4, b3, b2, b1, b0);
    CU_ASSERT_EQUAL(O_octetVersNaturel(resultat), 0);

    // Test pour 255 (Binaire : 11111111)
    b7 = 1, b6 = 1, b5 = 1, b4 = 1, b3 = 1, b2 = 1, b1 = 1, b0 = 1;
    resultat = O_creerOctet(b7, b6, b5, b4, b3, b2, b1, b0);
    CU_ASSERT_EQUAL(O_octetVersNaturel(resultat), 255);

    // Test pour 155 (Binaire : 10011011)
    b7 = 1, b6 = 0, b5 = 0, b4 = 1, b3 = 1, b2 = 0, b1 = 1, b0 = 1;
    resultat = O_creerOctet(b7, b6, b5, b4, b3, b2, b1, b0);
    CU_ASSERT_EQUAL(O_octetVersNaturel(resultat), 155);
}

void test_obtenir_ieme_bit(void) {
    // Test pour 155 (Binaire : 10011011)
    O_Octet octet = O_naturelVersOctet(155);

    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 7), bitA1);
    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 6), bitA0);
    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 5), bitA0);
    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 4), bitA1);
    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 3), bitA1);
    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 2), bitA0);
    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 1), bitA1);
    CU_ASSERT_EQUAL(O_obtenirIemeBit(octet, 0), bitA1);
}

void test_naturel_vers_octet(void) {
    unsigned char naturel = 42;
```

```
    O_Octet resultat = O_naturelVersOctet(naturel);
    CU_ASSERT_EQUAL(O_octetVersNaturel(resultat), naturel);
}

int main(int argc, char** argv) {
    /* initialisation du registre de tests */
    if (CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();

    /* ajout d'une suite de test pour octet.c */
    CU_pSuite pSuiteOctet = CU_add_suite("Test octet", init_suite_success,
    ↪  clean_suite_success);
    if (NULL == pSuiteOctet) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Ajout des tests à la suite octet */
    if ((NULL == CU_add_test(pSuiteOctet, "Création d'un octet", test_creer_octet))
    || (NULL == CU_add_test(pSuiteOctet, "Obtention du ième bit d'un octet",
    ↪  test_obtenir_ieme_bit))
    || (NULL == CU_add_test(pSuiteOctet, "Naturel vers octet", test_naturel_vers_octet))) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* ajout d'une suite de test pour statistiques.c */
    CU_pSuite pSuiteStatistiques = CU_add_suite("Test statistiques", init_suite_success,
    ↪  clean_suite_success);
    if (NULL == pSuiteStatistiques) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Ajout des tests à la suite statistiques */
    if ((NULL == CU_add_test(pSuiteStatistiques, "Création des statistiques aux occurences
    ↪  vides", test_statistiques_vides))
    || (NULL == CU_add_test(pSuiteStatistiques, "Incrementation de l'occurence d'un octet",
    ↪  test_statistiques_incrementees))
    || (NULL == CU_add_test(pSuiteStatistiques, "Fixer le nombre d'occurences d'un octet",
    ↪  test_statistiques_fixer_occurence))) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* ajout d'une suite de test pour codeBinaire.c */
    CU_pSuite pSuiteCodeBinaire = CU_add_suite("Test codeBinaire", init_suite_success,
    ↪  clean_suite_success);
    if (NULL == pSuiteCodeBinaire) {
        CU_cleanup_registry();
```

```
    return CU_get_error();
}

/* Ajout des tests à la suite codeBinaire */
if ((NULL == CU_add_test(pSuiteCodeBinaire, "Creation Code Binaire",
↪  test_creation_codebinaire))
|| (NULL == CU_add_test(pSuiteCodeBinaire, "Ajout d'un bit", test_ajout_bit))) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* ajout d'une suite de test pour fileDePrioriteDArbreDeHuffman.c */
CU_pSuite pSuiteFileDePriorite = CU_add_suite("Test fileDePrioriteDArbreDeHuffman",
↪  init_suite_success, clean_suite_success);
if (NULL == pSuiteFileDePriorite) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* Ajout des tests à la suite FileDePriorite*/
if ((NULL == CU_add_test(pSuiteFileDePriorite, "Création d'une File De Priorité vide",
↪  test_creation_filedePriorite_vide))
|| (NULL == CU_add_test(pSuiteFileDePriorite, "Enfiler des éléments dans une File De
↪  Priorité ", test_enfiler))
|| (NULL == CU_add_test(pSuiteFileDePriorite, "Obtenir un élément et défiler la File De
↪  Priorité", test_obtenir_element_et_defiler))) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* ajout d'une suite de test pour tableDeCodage.c */
CU_pSuite pTableDeCodage = CU_add_suite("Test tableDeCodage", init_suite_success,
↪  clean_suite_success);
if (NULL == pTableDeCodage) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* ajout des tests à la suite tableDeCodage */
if ((NULL == CU_add_test(pTableDeCodage, "Création d'une tableDeCodage 'vide'",
↪  test_creerTableCodage))
|| (NULL == CU_add_test(pTableDeCodage, "Vérifications multiples après l'ajout de 2
↪  CodeBinaire de tailles différentes dans la TableDeCodage", test_ajouterCodage))
|| (NULL == CU_add_test(pTableDeCodage, "Vérifications de la récupération d'un CodeBinaire
↪  après une unique insertion dans la TableDeCodage", test_octetVersCodeBinaire))) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* ajout d'une suite de test pour arbreDeHuffman.c */
```

```
    CU_pSuite pSuiteArbreDeHuffman = CU_add_suite("Test arbreDeHuffman",
    ↪  init_suite_success, clean_suite_success);
    if (NULL == pSuiteArbreDeHuffman) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Ajout des tests à la suite arbreDeHuffman */
    if ((NULL == CU_add_test(pSuiteArbreDeHuffman, "Creation d'un arbre de Huffman feuille à
    ↪  partir d'un octet et une occurence", test_creation_arbre_de_huffman_feuille))
    || (NULL == CU_add_test(pSuiteArbreDeHuffman, "Fusion de deux feuilles pour créer un
    ↪  arbre de Huffman", test_fusionner_ADH))) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Lancement des tests */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    printf("\n");
    CU_basic_show_failures(CU_get_failure_list());
    printf("\n\n");

    /* Nettoyage du registre */
    CU_cleanup_registry();
    return CU_get_error();
}
```

5.3.2 Tests des fonctions métier

```
#include <CUnit/Basic.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#include "codeBinaire.h"
#include "compression.c"
#include "decompression.c"
#include "fileDePrioriteDArbreDeHuffman.h"
#include "octet.h"
#include "statistiques.h"

int init_suite_success(void) {
    return 0;
}

int clean_suite_success(void) {
    return 0;
}

FILE *fichierTemporaireRempli() {
```

```
    FILE *tempFile = tmpfile();

    // Reprise des données du sujet pour effectuer les tests unitaires
    fprintf(tempFile, "BACFGABDDACEACG");

    return tempFile;
}

/* Tests construireArbreDeHuffman.c */

void test_file_de_priorite(void) {
    FILE *tempFile = fichierTemporaireRempli();

    S_Statistiques s;
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFile, &s, &taille);

    FDPAH_FileDePriorite fdp = CADH_construireFileDePriorite(s);


    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(FDPAH_obtenirElementEtDefiler(&fdp))),
    ↪ 'E');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(FDPAH_obtenirElementEtDefiler(&fdp))),
    ↪ 'F');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(FDPAH_obtenirElementEtDefiler(&fdp))),
    ↪ 'B');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(FDPAH_obtenirElementEtDefiler(&fdp))),
    ↪ 'D');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(FDPAH_obtenirElementEtDefiler(&fdp))),
    ↪ 'G');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(FDPAH_obtenirElementEtDefiler(&fdp))),
    ↪ 'C');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(FDPAH_obtenirElementEtDefiler(&fdp))),
    ↪ 'A');

    fclose(tempFile);
}

void test_arbre_de_huffman(void) {
    FILE *tempFile = fichierTemporaireRempli();

    S_Statistiques s;
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFile, &s, &taille);
```

```
    ADH_ArbreDeHuffman a = CADH_construireArbreDeHuffman(s);


    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(ADH_obtenirFilsGauche(ADH_obtenirFils
    ↪ 'C');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(ADH_obtenirFilsDroit(ADH_obtenirFilsGa
    ↪ 'A');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(ADH_obtenirFilsGauche(ADH_obtenirFils
    ↪ 'B');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(ADH_obtenirFilsDroit(ADH_obtenirFilsGa
    ↪ 'D');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(ADH_obtenirFilsGauche(ADH_obtenirFils
    ↪ 'G');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(ADH_obtenirFilsGauche(ADH_obtenirFils
    ↪ 'E');

    ↪ CU_ASSERT_EQUAL(O_octetVersNaturel(ADH_obtenirOctet(ADH_obtenirFilsDroit(ADH_obtenirFilsDr
    ↪ 'F');

    fclose(tempFile);
}

/* Tests compression.c */

void test_obtenir_statistiques(void) {
    FILE *tempFile = fichierTemporaireRempli();

    S_Statistiques s;
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFile, &s, &taille);

    CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet('A')), 4);
    CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet('B')), 2);
    CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet('C')), 3);
    CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet('D')), 2);
    CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet('E')), 1);
    CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet('F')), 1);
    CU_ASSERT_EQUAL(S_obtenirOccurence(s, O_naturelVersOctet('G')), 2);

    fclose(tempFile);
}

void test_obtenir_taille_fichier(void) {
    FILE *tempFile = fichierTemporaireRempli();

    S_Statistiques s;
```

```
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFile, &s, &taille);

    CU_ASSERT_EQUAL(taille, 4 + 2 + 3 + 2 + 1 + 1 + 2);

    fclose(tempFile);
}

void test_table_de_codage(void) {
    FILE *tempFile = fichierTemporaireRempli();

    S_Statistiques s;
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFile, &s, &taille);

    ADH_ArbreDeHuffman a = CADH_construireArbreDeHuffman(s);

    TDC_TableDeCodage tdc = C_obtenirTableDeCodage(a);

    CB_CodeBinaire cb;

    // On effectue un test sur 3 octets
    cb = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet('A'));
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 0), bitA0);
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 1), bitA1);

    cb = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet('C'));
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 0), bitA0);
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 1), bitA0);

    cb = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet('E'));
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 0), bitA1);
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 1), bitA1);
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 2), bitA1);
    CU_ASSERT_EQUAL(CB_obtenirIemeBit(cb, 3), bitA0);

    fclose(tempFile);
}

void test_code_binaire_8_bits_vers_octet(void) {
    unsigned short i;

    O_Octet o = O_naturelVersOctet('K');

    CB_CodeBinaire cb = CB_creerCodeBinaire(O_obtenirIemeBit(o, 0));
    for (i = 1; i < MAX_BITS; i++)
        CB_ajouterBit(&cb, O_obtenirIemeBit(o, i));

    O_Octet otest = C_codeBinaireEnOctet(cb);

    CU_ASSERT_EQUAL(O_octetVersNaturel(o), O_octetVersNaturel(otest));
```

```
}

void test_ecrire_identifiant(void) {
    FILE *tempFile = tmpfile();

    C_ecrireIdentifiant(tempFile);

    rewind(tempFile);
    unsigned short identifiant_lue;
    unsigned short identifiant_attendue = IDENTIFIANT;

    CU_ASSERT_EQUAL(fread(&identifiant_lue, sizeof(unsigned short), 1, tempFile), 1);
    CU_ASSERT_EQUAL(identifiant_lue, identifiant_attendue);
    fclose(tempFile);
}

void test_ecrire_taille_fichier(void) {
    FILE *tempFileEntree = fichierTemporaireRempli();

    S_Statistiques s;
    unsigned long long taille;

    C_obtenirStatistiquesEtTailleFichier(tempFileEntree, &s, &taille);
    fclose(tempFileEntree);

    FILE *tempFileSortie = tmpfile();

    C_ecrireTailleFichier(tempFileSortie, taille);

    rewind(tempFileSortie);
    unsigned long long taille_lue;

    CU_ASSERT_EQUAL(fread(&taille_lue, sizeof(unsigned long long), 1, tempFileSortie), 1);
    CU_ASSERT_EQUAL(taille_lue, taille);

    fclose(tempFileSortie);
}

void test_ecrire_statistiques(void) {
    FILE *tempFileEntree = fichierTemporaireRempli();

    S_Statistiques s_entree;
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFileEntree, &s_entree, &taille);

    fclose(tempFileEntree);

    FILE *tempFileSortie = tmpfile();

    C_ecrireStatistiques(tempFileSortie, s_entree);
```

```
rewind(tempFileSortie);
unsigned long occurence;
unsigned char octet;
size_t result;

result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(occurence, 4);

result = fread(&octet, sizeof(unsigned char), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(octet, 'A');

result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(occurence, 2);

result = fread(&octet, sizeof(unsigned char), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(octet, 'B');

result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(occurence, 3);

result = fread(&octet, sizeof(unsigned char), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(octet, 'C');

result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(occurence, 2);

result = fread(&octet, sizeof(unsigned char), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(octet, 'D');

result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(occurence, 1);

result = fread(&octet, sizeof(unsigned char), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(octet, 'E');

result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
CU_ASSERT_EQUAL(occurence, 1);

result = fread(&octet, sizeof(unsigned char), 1, tempFileSortie);
CU_ASSERT_EQUAL_FATAL(result, 1);
```

```
    CU_ASSERT_EQUAL(octet, 'F');

    result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(occurence, 2);

    result = fread(&octet, sizeof(unsigned char), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(octet, 'G');

    result = fread(&occurence, sizeof(unsigned long), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(occurence, 0);

    fclose(tempFileSortie);
}

void test_concatener_codes_binaires(void) {
    FILE *tempFileEntree = fichierTemporaireRempli();

    unsigned short i;
    S_Statistiques s;
    unsigned long long longueur;
    C_obtenirStatistiquesEtTailleFichier(tempFileEntree, &s, &longueur);
    ADH_ArbreDeHuffman a = CADH_construireArbreDeHuffman(s);
    TDC_TableDeCodage tdc = C_obtenirTableDeCodage(a);

    FILE *tempFileSortie = tmpfile();

    CB_CodeBinaire cbTemp = CB_creerCodeBinaire(bitA0);
    for (i = 1; i < MAX_BITS; i++)
        CB_ajouterBit(&cbTemp, bitA0);

    // Boucle d'encodage
    CB_CodeBinaire cb;
    short o;
    while ((o = fgetc(tempFileEntree)) != EOF) {
        cb = TDC_octetVersCodeBinaire(tdc, O_naturelVersOctet(o));
        C_concatenerCodeBinaireDansFichier(tempFileSortie, &cbTemp, cb);
    }

    rewind(tempFileSortie);

    unsigned char octet1, octet2;
    size_t result;
    unsigned char otest1 = O_octetVersNaturel(O_creerOctet(bitA1, bitA0, bitA0, bitA0, bitA1,
    ↪  bitA0, bitA0, bitA1));
    unsigned char otest2 = O_octetVersNaturel(O_creerOctet(bitA1, bitA1, bitA1, bitA1, bitA1,
    ↪  bitA0, bitA0, bitA1));

    result = fread(&octet1, sizeof(unsigned char), 1, tempFileSortie);
```

```
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(octet1, otest1);  // 10001001 en binaire

    result = fread(&octet2, sizeof(unsigned char), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(octet2, otest2);  // 11111001 en binaire

    fclose(tempFileSortie);
    fclose(tempFileEntree);
}

void test_encoder(void) {
    S_Statistiques s;
    unsigned long long taille;
    FILE *tempFileEntree = fichierTemporaireRempli();

    rewind(tempFileEntree);
    FILE *tempFileSortie = tmpfile();
    rewind(tempFileSortie);
    C_obtenirStatistiquesEtTailleFichier(tempFileEntree, &s, &taille);

    if (taille > 0) {
        ADH_ArbreDeHuffman a = CADH_construireArbreDeHuffman(s);
        if (!ADH_estUneFeuille(a))  // Cas particulier d'un fichier contenant un seul octet (présent
        ↪  plusieurs fois ou non)
            C_encoder(tempFileEntree, tempFileSortie, C_obtenirTableDeCodage(a));
        ADH_liberer(a);
    }

    rewind(tempFileSortie);
    rewind(tempFileEntree);

    unsigned char octet1, octet2, octet3, octet4, octet5;
    size_t result;

    unsigned char otest1 = O_octetVersNaturel(O_creerOctet(bitA1, bitA0, bitA0, bitA1, bitA0,
    ↪  bitA0, bitA0, bitA1));
    result = fread(&octet1, sizeof(unsigned char), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(octet1, otest1);  // 10001001

    unsigned char otest2 = O_octetVersNaturel(O_creerOctet(bitA1, bitA0, bitA0, bitA1, bitA1,
    ↪  bitA1, bitA1, bitA1));
    result = fread(&octet2, sizeof(unsigned char), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(octet2, otest2);  // 11111001

    unsigned char otest3 = O_octetVersNaturel(O_creerOctet(bitA0, bitA1, bitA1, bitA0, bitA1,
    ↪  bitA0, bitA0, bitA1));
    result = fread(&octet3, sizeof(unsigned char), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
```

```
    CU_ASSERT_EQUAL(octet3, otest3);  // 10010110

    unsigned char otest4 = O_octetVersNaturel(O_creerOctet(bitA1, bitA1, bitA1, bitA0, bitA0,
    ↪  bitA1, bitA0, bitA1));
    result = fread(&octet4, sizeof(unsigned char), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(octet4, otest4);  // 10100111

    unsigned char otest5 = O_octetVersNaturel(O_creerOctet(bitA0, bitA1, bitA1, bitA0, bitA0,
    ↪  bitA1, bitA0, bitA0));
    result = fread(&octet5, sizeof(unsigned char), 1, tempFileSortie);
    CU_ASSERT_EQUAL_FATAL(result, 1);
    CU_ASSERT_EQUAL(octet5, otest5);  // 00100110

    fclose(tempFileSortie);
    fclose(tempFileEntree);
}

/* Tests decompression.c */

void test_seDeplacerDansLArbre(void) {
    FILE *tempFileEntree = fichierTemporaireRempli();
    S_Statistiques s;
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFileEntree, &s, &taille);
    ADH_ArbreDeHuffman abh = CADH_construireArbreDeHuffman(s);
    // On vérifie arbitrairement si on arrive à retrouver notre octet tout à gauche, celui tout à droite
    ↪  et un dernier entre les deux selon l'exemple du sujet
    // L'octet 'C' se situe 2 cran à gauche
    ADH_ArbreDeHuffman abhTest = abh;
    for (unsigned int i = 0; i < 2; i++) {
        D_seDeplacerDansLArbre(bitA0, &abhTest);
    }
    CU_ASSERT(O_octetVersNaturel(ADH_obtenirOctet(abhTest)) == 'C');

    // L'octet 'F' se situe 4 crans à droite
    abhTest = abh;
    for (unsigned int i = 0; i < 4; i++) {
        D_seDeplacerDansLArbre(bitA1, &abhTest);
    }
    CU_ASSERT(O_octetVersNaturel(ADH_obtenirOctet(abhTest)) == 'F');

    // L'octet 'D' se situe 1 cran à droite, puis 1 à gauche et enfin 1 à droite
    abhTest = abh;
    D_seDeplacerDansLArbre(bitA1, &abhTest);
    D_seDeplacerDansLArbre(bitA0, &abhTest);
    D_seDeplacerDansLArbre(bitA1, &abhTest);
    CU_ASSERT(O_octetVersNaturel(ADH_obtenirOctet(abhTest)) == 'D');

    fclose(tempFileEntree);
}
```

```c
void test_lire_statistiques(void) {
    FILE *tempFileEntree = fichierTemporaireRempli();

    S_Statistiques s_entree;
    unsigned long long taille;
    C_obtenirStatistiquesEtTailleFichier(tempFileEntree, &s_entree, &taille);

    fclose(tempFileEntree);

    FILE *tempFileSortie = tmpfile();

    C_ecrireStatistiques(tempFileSortie, s_entree);

    rewind(tempFileSortie);

    S_Statistiques s_lu;
    D_lireStatistiques(tempFileSortie, &s_lu);

    for (unsigned long int o = 0; o < 256; ++o) {
        unsigned long occurence = S_obtenirOccurence(s_lu, O_naturelVersOctet(o));
        switch (o) {
            case 'A':
                CU_ASSERT_EQUAL(occurence, 4);
                break;
            case 'B':
                CU_ASSERT_EQUAL(occurence, 2);
                break;
            case 'C':
                CU_ASSERT_EQUAL(occurence, 3);
                break;
            case 'D':
                CU_ASSERT_EQUAL(occurence, 2);
                break;
            case 'E':
                CU_ASSERT_EQUAL(occurence, 1);
                break;
            case 'F':
                CU_ASSERT_EQUAL(occurence, 1);
                break;
            case 'G':
                CU_ASSERT_EQUAL(occurence, 2);
                break;
            default:
                CU_ASSERT_EQUAL(occurence, 0);
                break;
        }
    }

    fclose(tempFileSortie);
}
```

```c
void test_decoder(void) {
    FILE *fichierTest = fichierTemporaireRempli();
    S_Statistiques s;
    unsigned long long longueur;
    C_obtenirStatistiquesEtTailleFichier(fichierTest, &s, &longueur);
    ADH_ArbreDeHuffman a = CADH_construireArbreDeHuffman(s);

    // Création du fichier encoder de l'exemple
    FILE *fichierTestEncode = tmpfile();
    TDC_TableDeCodage tdc = C_obtenirTableDeCodage(a);
    rewind(fichierTest);
    C_encoder(fichierTest, fichierTestEncode, tdc);

    // On décode ce fichier fraichement encodé
    FILE *fichierTestDecode = tmpfile();
    rewind(fichierTestEncode);
    D_decoder(a, longueur, fichierTestEncode, fichierTestDecode);
    fclose(fichierTestEncode);

    // On regarde si tous les octets entre le fichier original et le fichier decoder sont égaux
    rewind(fichierTest);
    rewind(fichierTestDecode);
    for (unsigned int i = 1; i <= longueur; i++) {
        unsigned char octetActuelFichierTest;
        size_t nbBlocsLus = fread(&octetActuelFichierTest, sizeof(unsigned char), 1, fichierTest);
        if (nbBlocsLus < 1) {
            printf("Erreur testsFonctionsMetier.c : fin du fichier atteinte de manière innatendue ou
            ↪  erreur de la fonction fread \n");
            exit(EXIT_FAILURE);
        }

        unsigned char octetActuelFichierTestDecode;
        nbBlocsLus = fread(&octetActuelFichierTestDecode, sizeof(unsigned char), 1,
        ↪  fichierTestDecode);
        if (nbBlocsLus < 1) {
            printf("Erreur testsFonctionsMetier.c : fin du fichier atteinte de manière innatendue ou
            ↪  erreur de la fonction fread \n");
            exit(EXIT_FAILURE);
        }

        CU_ASSERT_EQUAL(octetActuelFichierTest, octetActuelFichierTestDecode);
    }

    fclose(fichierTest);
    fclose(fichierTestDecode);
}

int main(int argc, char **argv) {
    /* initialisation du registre de tests */
    if (CUE_SUCCESS != CU_initialize_registry())
```

```
    return CU_get_error();

/* ajout d'une suite de test pour construireArbreDeHuffman.c*/
CU_pSuite pSuiteConstruireArbreDeHuffman = CU_add_suite("Test construction de l'Arbre de
↪  Huffman", init_suite_success, clean_suite_success);
if (NULL == pSuiteConstruireArbreDeHuffman) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* Ajout des tests à la suite compression */
if ((NULL == CU_add_test(pSuiteConstruireArbreDeHuffman, "Construction de la file de
↪  priorité à partir des statistiques", test_file_de_priorite))
|| (NULL == CU_add_test(pSuiteConstruireArbreDeHuffman, "Construction de l'arbre de
↪  Huffman à partir des statistiques", test_arbre_de_huffman))) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* ajout d'une suite de test pour compression.c */
CU_pSuite pSuiteCompression = CU_add_suite("Test compression", init_suite_success,
↪  clean_suite_success);
if (NULL == pSuiteCompression) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* Ajout des tests à la suite compression */
if ((NULL == CU_add_test(pSuiteCompression, "Obtention des statistiques d'un fichier",
↪  test_obtenir_statistiques))
|| (NULL == CU_add_test(pSuiteCompression, "Obtention de la taille d'un fichier",
↪  test_obtenir_taille_fichier))
|| (NULL == CU_add_test(pSuiteCompression, "Obtention de la table de codage à partir de
↪  l'arbre de huffman", test_table_de_codage))
|| (NULL == CU_add_test(pSuiteCompression, "Conversion d'un code binaire de 8 bits vers un
↪  octet", test_code_binaire_8_bits_vers_octet))
|| (NULL == CU_add_test(pSuiteCompression, "ecrire un identifiant dans un fichier ",
↪  test_ecrire_identifiant))
|| (NULL == CU_add_test(pSuiteCompression, "ecrire la taille du fichier dans un fichier",
↪  test_ecrire_taille_fichier))
|| (NULL == CU_add_test(pSuiteCompression, "ecrire les statistique du fichier dans un
↪  fichier", test_ecrire_statistiques))
|| (NULL == CU_add_test(pSuiteCompression, "encodage d'un fichier et vérification que la
↪  concatenation fonctionne", test_encoder))) {
    CU_cleanup_registry();
    return CU_get_error();
}

/* ajout d'une suite de test pour decompression.c */
CU_pSuite pSuiteDecompression = CU_add_suite("Test decompression", init_suite_success,
↪  clean_suite_success);
```

```
    if (NULL == pSuiteDecompression) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Ajout des tests à la suite decompression */
    if ((NULL == CU_add_test(pSuiteDecompression, "Lecture des statistiques",
    ↪  test_lire_statistiques))
    || (NULL == CU_add_test(pSuiteDecompression, "3 tests arbitraires pour
    ↪  D_seDeplacerDansLArbre", test_seDeplacerDansLArbre))
    || (NULL == CU_add_test(pSuiteDecompression, "Decodage d'un fichier", test_decoder))) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Lancement des tests */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    printf("\n");
    CU_basic_show_failures(CU_get_failure_list());
    printf("\n\n");

    /* Nettoyage du registre */
    CU_cleanup_registry();
    return CU_get_error();
}
```

Chapitre 6

Distribution des tâches

6.1 Tableaux de distribution des tâches

		Taoba	Thomas	Ali	Mathis
Analyse (TAD)	CodeBinaire			×	
	Octet				×
	Statistiques	×			
	ArbreDeHuffman		×		
	FileDePriorité		×		
	TableDeCodage				×
CP	Code Binaire				×
	Octet		×		
	Statistiques		×		
	ArbreDeHuffman	×			
	FileDePriorité				×
	TableDeCodage			×	
CD	CodeBinaire	×			
	Octet	×			
	Statistiques				×
	ArbreDeHuffman			×	×
	FileDePriorité			×	
	TableDeCodage		×		
Dev	CodeBinaire		×		
	Octet			×	
	Statistiques		×		
	ArbreDeHuffman			×	×
	FileDePriorité				×
	TableDeCodage	×			
Tests unitaires	CodeBinaire		×		
	Octet			×	
	Statistiques		×		
	ArbreDeHuffman	×			
	FileDePriorité	×			
	TableDeCodage				×

Table 6.1 – Tableau des tâches des TADs

		Taoba	Thomas	Ali	Mathis
Analyse	obtenirStatistiqueEtTailleFichier			×	×
	ecrire{Statistiques,Identifiant,Taille}			×	×
	construireFileDePriorité	×	×	×	×
	construireArbreDeHuffman	×	×	×	×
	codeBinaireEnOctet			×	×
	obtenirTableDeCodage			×	×
	concatenerCodeBinaireDansFichier			×	×
	encoder			×	×
	compresser			×	×
CP	obtenirStatistiqueEtTailleFichier	×			
	ecrire{Statistiques,Identifiant,Taille}		×		
	construireFileDePriorité	×			
	construireArbreDeHuffman			×	
	obtenirTableDeCodage	×			
	codeBinaireEnOctet		×		
	concatenerCodeBinaireDansFichier		×		
	encoder		×		
	compresser	×			
CD	obtenirStatistiqueEtTailleFichier		×		
	ecrire{Statistiques,Identifiant,Taille}	×			
	construireFileDePriorité		×		
	construireArbreDeHuffman			×	
	obtenirTableDeCodage		×		
	codeBinaireEnOctet	×			
	concatenerCodeBinaireDansFichier	×	×		
	encoder	×			
	compresser		×		
Dev	obtenirStatistiqueEtTailleFichier	×			
	ecrire{Statistiques,Identifiant,Taille}		×		
	construireFileDePriorité	×			
	construireArbreDeHuffman				×
	obtenirTableDeCodage	×			
	codeBinaireEnOctet		×		
	concatenerCodeBinaireDansFichier		×		
	encoder		×		
	compresser	×			
Tests Unitaires	obtenirStatistiqueEtTailleFichier		×		
	ecrire{Statistiques,Identifiant,Taille}	×			
	construireFileDePriorité		×		
	construireArbreDeHuffman		×		
	obtenirTableDeCodage		×		
	codeBinaireEnOctet	×			
	concatenerCodeBinaireDansFichier	×			
	encoder	×			
	compresser		×		

Table 6.2 – Tableau des tâches de la compression

		Taoba	Thomas	Ali	Mathis
Analyse	lireStatistiques	×	×		
	construireFileDePriorité	×	×	×	×
	construireArbreDeHuffman	×	×	×	×
	seDeplacerDansLArbre	×	×		
	decoder	×	×		
	decompresser	×	×		
CP	lireStatistiques			×	
	construireFileDePriorité	×			
	construireArbreDeHuffman			×	
	seDeplacerDansLArbre				×
	decoder				×
	decompresser			×	
CD	lireStatistiques				×
	construireFileDePriorité		×		
	construireArbreDeHuffman				×
	seDeplacerDansLArbre			×	
	decoder			×	
	decompresser				×
Dev	lireStatistiques			×	
	construireFileDePriorité	×			
	construireArbreDeHuffman			×	
	seDeplacerDansLArbre				×
	decoder				×
	decompresser				×
Tests Unitaires	lireStatistiques			×	
	construireFileDePriorité		×		
	construireArbreDeHuffman		×		
	seDeplacerDansLArbre				×
	decoder				×
	decompresser			×	

Table 6.3 – Tableau des tâches de la décompression

Chapitre 7

Conclusion

Dans ce projet, nous avons élaboré un algorithme de compression sans perte basé sur l'Arbre de Huffman, en suivant une conception en quatre phases :

— La première phase a consisté en la conception des TADs et des analyses descendantes nécessaires à la mise en place de notre compresseur Huffman.
— Ensuite, nous avons réalisé la conception préliminaire de nos fonctions et procédures, jetant ainsi les bases de l'implémentation.
— La troisième phase a été consacrée à la conception détaillée, où nous avons approfondi chaque aspect de l'algorithme, clarifiant les spécifications et les interactions entre les différentes parties du code.
— Enfin, nous avons procédé à l'implémentation du code en langage C et à la réalisation des tests unitaires pour évaluer le bon fonctionnement des algorithmes.

Les résultats des tests, principalement réalisés sur des fichiers textes, démontrent l'efficacité de notre algorithme en termes de réduction de la taille des données, tout en préservant l'intégrité des informations d'origine.

L'utilisation d'une compression sans perte de données pour les fichiers texte est cruciale, car elle garantit la conservation du sens du texte d'origine. Nous aurions également pu explorer des méthodes de compression par perte de données et discuter de leur utilisation sur des fichiers audio ou images sur lesquels notre algorithme ne fonctionne pas, étant des fichiers déjà compressés.

Le travail de groupe a été essentiel à la réussite de ce projet. Nous avons pu partager les tâches et les responsabilités, bénéficier des compétences et des connaissances des autres membres du groupe, et résoudre les problèmes de manière collective.

Concernant les perspectives d'amélioration, notre algorithme de compression pourrait être optimisé de différentes manières. Par exemple, l'utilisation d'un autre algorithme pour l'encodage des données ou l'exploration de techniques de compression plus avancées telles que la compression par blocs ou la compression par transformation.