ROLLET Samuel
SALLE Jennifer

Compression de Données - Algorithme de Huffman
Document de Conception

Projet d'Algorithmique et Structure des Données

SOMMAIRE

1. Domaine d'application

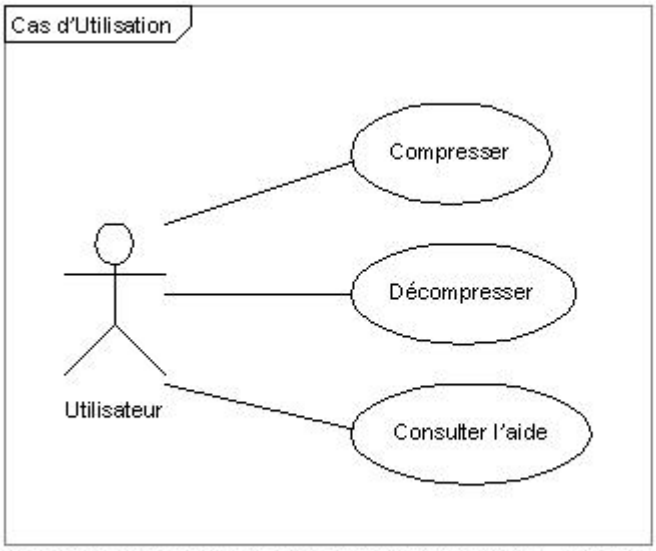
1.1 Objectifs du système

L'objectif est de pouvoir effectuer la compression et la décompression de fichiers en utilisant la méthode de compression de Huffman. Le programme offrira aussi une aide aux utilisateurs.

1.2 Interface utilisateur, logiciels et matériels

Le programme *jszip* sera accessible en ligne de commande depuis une interface *shell* sous Linux il n'y aura donc pas d'interface graphique. L'utilisateur devra taper la commande *jszip* suivis d'options pour effectuer les différentes tâches offertes par ce programme. Il n'y a pas de restrictions matérielles particulières à signaler.

Les différentes possibilités offertes aux utilisateurs sont résumées dans le schéma suivant.



Ces différentes possibilités seront accessibles par l'utilisateur en tapant les options suivantes à la suite de *jszip* dans une console :

- **-h** : affichage de l'aide.
- **-c** *liste des fichiers à compresser* ***-o*** *nom du fichier compressé*

  Les fichiers dont le nom est donné après le -c sont les fichiers que l'utilisateur souhaite compresser.

  Le nom donné après le -o est le nom de l'archive que l'utilisateur souhaite créer; si ce nom n'est pas terminé par l'extension *.jszip* celle-ci sera ajoutée automatiquement.

- **-d** *nom de l'archive à décompresser*

  Le nom du fichier placé après -d doit être une archive *.jszip* valide, qui sera décompressée dans un sous dossier portant le même nom que l'archive. Si le nom de l'archive n'est pas terminé par l'extension *.jszip* celle-ci sera ajoutée automatiquement.

Des messages d'erreur seront affichés dans les cas suivant:

- absence de noms de fichiers à compresser ou du nom de l'archive lors d'une compression.
- inexistence des fichiers à compresser.
- absence du nom d'une archive lors d'une décompression.
- archive corrompue ou mauvaise extension de celle-ci.

1.3 Contraintes générales de conception

L'ensemble des programmes de notre logiciel sont codés en langage C et doivent respecter les règles de codage contenues dans le fichier Qualité.txt. De plus, notre logiciel ne peut être compilé et exécuté que sous un environnement Linux. Nous utilisons le compilateur gcc avec les options *-Wall -Werror* afin de considérer tous les warning comme des erreurs et d'éviter des erreurs classiques dans le code. Enfin, le code développé sera vérifié en utilisant LcLint, un outil de vérification des programmes en C contre les erreurs de codage et les failles de sécurité.

2. Documents de référence

Les documents de références sont les suivants :

- Règles de Codage en C - ESIL
- Compression de Données - Algorithme de Huffman - ESIL
- Huffman Coding - Wikipedia
- Heap - Wikipedia
- Splint Manual

3. Conception générale

3.1 Langage utilisé

Le langage utilisé est le langage C comme définit dans le sujet. Nous respecteront les règles de codage définit dans le document *Règles de Codage* qui nous a été fournit.

3.2 Architecture générale et interfaces détaillées des modules

Il existe trois modules principaux qui correspondent aux fonctions offertes par le programme: Compression, Décompression, et Aide.

Les modules Compressions et Décompression utilisent d'autres modules pour effectuer leurs tâches.

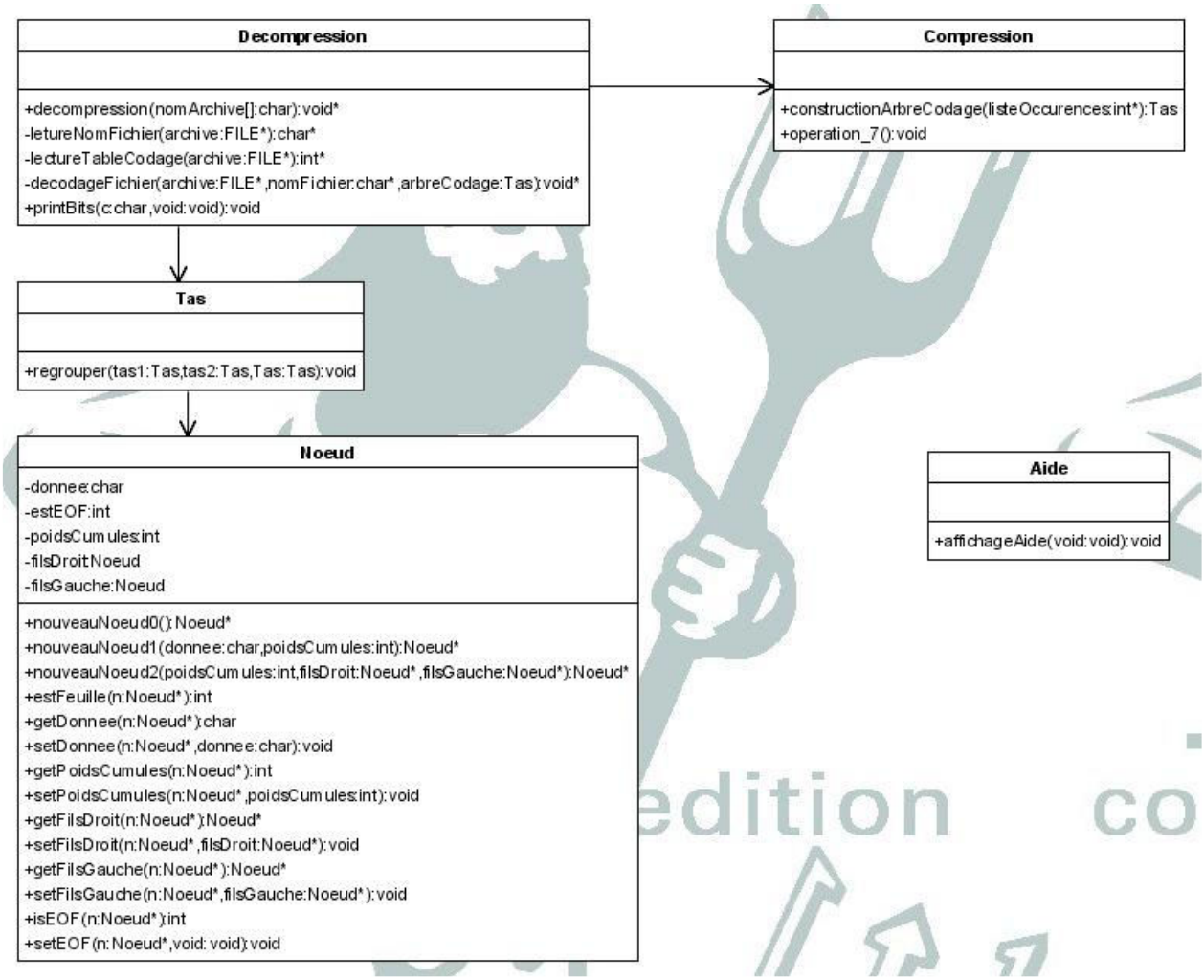
Dans les diffèrent schémas que nous vous proposons, un cube représente un module qui correspond à un fichier *.c et *.h. Une flèche d'un module A vers un module B signifie que le module A utilise des fonctions exportés par le module B. De plus, les fonctions publics sont repérées par un "+" et les fonctions privé par un "-" avant le nom de la fonction.

Enfin, la description des modules et des fonctions a été générée par *Doxygen* qui utilise les commentaires imposés dans le code pour produire une documentation.

3.2.1 La compression

Compression
-lectureFichier(nomFichierSource[]:char):int*
+constructionArbreCodage(listeOccurences:int*):Tas
-calculeCode(codesTas:TableCodage*):void
-parcoursArbre(codesTas,listeCodesTas:TableCodage*,codeCouran:CodeCaractere*,position:int):void
-ecritureArchive(archive:FILE*,nomFichierSource[]:char,listeCodesTableCodage*):void
+compression(nbFichiers:int,nomsFichiers[]:char*,nomArchive:char*):int
Buffer
-tampon:char
-code:char
-libre:int
-tamponUtilise:int
+initBuffer(Buffer* courant:Buffer* courant):void
+shift(Buffer* courant:Buffer* courant):int
TableCodage
-tableau:CodeCaractere*
CodeCaractere
-code:char*
-longueurCode:int
+nouveauCodeCaractere0():CodeCaractere*
+nouveauCodeCaractere1():void
+setCode(courant:CodeCaractere*,code:char*):void
+setLongueurCode(courant:CodeCaractere*,longueurCode:int):void
+setLongueurCodeMax(courant:CodeCaractere*,longueurCode:int):void
+copieCode(source:CodeCaractere*,destination:CodeCaractere*):void
+getCode(courant:CodeCaractere*):char*
+getLongueurCode(courant:CodeCaractere*):int
+setNextBite(courant:CodeCaractere*,nouveauBit:int):void
+setBit(courant:CodeCaractere*,position:int,nouveauBit:int):void
+afficherCode(courant:CodeCaractere*):void
ListeTriee
+creerListeVide():ListeTriee
+ajouterMaillon(listeCourante:ListeTriee,nouveauMaillon:Maillon*):ListeTriee
+supprimerTete(listeCourante:ListeTriee):ListeTriee
Tas
+regrouper(tas1:Tas,tas2:Tas,Tas:Tas):void
Noeud
-donnee:char
-estEOF:int
-poidsCumules:int
-filsDroit:Noeud
-filsGauche:Noeud
+nouveauNoeud0():Noeud*
+nouveauNoeud1(donnee:char,poidsCumules:int):Noeud*
+nouveauNoeud2(poidsCumules:int,filsDroit:Noeud*,filsGauche:Noeud*):Noeud*
+estFeuille(n:Noeud*):int
+getDonnee(n:Noeud*):char
+setDonnee(n:Noeud*,donnee:char):void
+getPoidsCumules(n:Noeud*):int
+setPoidsCumules(n:Noeud*,poidsCumules:int):void
+getFilsDroit(n:Noeud*):Noeud*
+setFilsDroit(n:Noeud*,filsDroit:Noeud*):void
+getFilsGauche(n:Noeud*):Noeud*
+setFilsGauche(n:Noeud*,filsGauche:Noeud*):void
+isEOF(n:Noeud*):int
+setEOF(n:Noeud*,void:void):void
Maillon
-suivant:Maillon*
-arbre:Tas
+nouveauMaillon0():Maillon*
+nouveauMaillon1(arbre:Tas):Maillon*
+detruire(courant:Maillon*):void
+getPoid(courant:Maillon*):int
+getSuivant(courant:Maillon*):Maillon*
+setSuivant(courant:Maillon*,suivant:Maillon*):int
+getArbre(courant:Maillon*):Noeud*
+setArbre(courant:Maillon*,noeud:Tas):void

3.2.2 La décompression et l'aide



3.3 Structure des fichiers et données globales

Notre logiciel ne comporte aucune donnée globale. Toutes les données sont passées en paramètre aux fonctions.

Le fichier d'entête *InfoComunes.h* regroupe les messages d'informations, les *magics numbers* et le nombre de caractère de la table ASCII.

Il y a vingt-deux fichiers et le Makefile :

- Buffer.c
- Buffer.h
- CodeCaractere.c
- CodeCaractere.h
- Compression.c
- Compression.h
- ListeTriee.c
- ListeTriee.h
- Maillon.c
- Maillon.h
- Decompression.c
- Decompression.h
- main.c
- Noeud.c
- Noeud.h
- TableCodage.c
- TableCodage.h
- Tas.c
- Tas.h
- InfosComunes.h
- Aide.c
- Aide.h

Les fichiers sources *.c débutent par une entête (nom du module, auteurs et description) respectant les normes d'écriture de commentaire du *Doxygen*, indiquent les inclusions et si nécessaire les déclarations des fonctions PRIVATE et des variables. Les fonctions PUBLIC et PRIVATE sont ensuite codées.

Les fichiers *.h, quant à eux, contiennent les prototypes des fonctions PUBLIC du module correspondant.

3.4 Stratégie de traitement des erreurs

Les erreurs du logiciel peuvent venir d'une mauvaise saisie de l'utilisateur (argument invalide passé au programme). Afin de ne pas perturber le bon déroulement du programme, nous avons alors décidé d'afficher un message d'erreur à chaque mauvaise saisie et nous conseillons à l'utilisateur de consulter l'aide.

3.5 Justification du choix d'architecture, des modules et du langage

Le choix du langage nous a été imposé pour ce projet. Il s'agit du langage de programmation C.

Le programme se découpe en trois modules principaux correspondant aux fonctions offertes par le programme : *compression*, *compression* et *aide*.

Les modules de *compression* et de *compression* utilisent d'autres modules afin de

manipuler les données et de stocker les informations nécessaires à la gestion des fichiers compressés.

Les modules *Tas* et *Nœud* offrent les fonctionnalités nécessaires au stockage de l'arbre de codage.

Les modules *TableCodage* et *CodeCaractere* offrent les fonctionnalités nécessaires au stockage des codes de Huffman associés à chaque caractère.

Les modules *ListeTriee* et *Maillon* permettent la gestion d'une liste d'arbre triée par ordre décroissant suivant le poids de la racine de chaque arbre.

Le module *Buffer* offre les fonctionnalités nécessaires à l'écriture des codes caractères dans un fichier octet par octet.

3.6 Structure des fichiers d'archives

Une fois la compression effectuée, les fichiers archives débutent par le *magic number*. Puis, pour chaque fichier texte compressé, le nom du fichier texte est indiqué, suivit de sa table d'occurrence et du texte compressé.

De plus, la fin de la table d'occurrence est repéré par "0" pour pouvoir identifier la fin de la table d'occurrence lors de la lecture de celle ci pendant la décompression de l'archive.

Disposition :

Magic number
Nom.txt
Table d'occurrence
Texte compressé
Nom2.txt
Table d'occurrence
Texte compressé

Exemple :

```
9014282
test2.txt
10 4 45 3 46 2 97 4 98 1 99 2 100 1 101 1 102 3 103 1 104 1 105
1 106 2 107 1 108 1 0 0
m¾!P»vˆ&`ù¦Ušt_
manual.txt
10 855 32 8980 33 12 34 49 35 1 39 33 40 91 41 93 42 8 43 27 44
199 45 1111 46 423 47 105 48 119 49 77 50 54 51 72 52 32 53 39
54 48 55 15 56 38 57 32 58 120 59 3 60 17 61 417 62 22 63 4 ……
¹z€@_ê€±zL__¤_UUUUUUUUUUUUV+Ðš:,5Œ5½ñÇ_qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_
qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_qÇ_ouKÔ__/T_Ô_·BéØi!¸®+Ð¥˜ÏD¼ÀI;……
```

4. Conception détaillée des modules (NDT)

4.1 Fonction Main

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Compression.h"
#include "Decompression.h"
#include "InfosComunes.h"
```

Fonctions

1. PUBLIC int **main** (int argc, char *argv[])

Description détaillée

Point d'entrée du programme. Projet jsZip: Compression et décompression de fichiers texte grâce a l'algorithme de Huffman.

Analyse des arguments, lance la compression, la décompression ou l'affichage de l'aide.

Description des fonctions

PUBLIC int main (int *argc*, char * *argv*[])

Point d'entrée du programme

Paramètres :

argc nombre d'arguments
argv arguments

Returns:

EXIT_SUCCESS si le programme c'est termine correctement. EXIT_FAILURE sinon.

Références compression().

4.2 Module Compression

4.2.1 Description des structures de données

a) Buffer

```
#include <Buffer.h>
```

Champs de données

1. **ListeBits codeTampon**
2. int **libre**
3. int **tamponUtilise**

Description détaillée

Structure du **Buffer**. Contient les informations nécessaires à la gestion des 16bits du buffer.

Détaille des champs

ListeBits Buffer::codeTampon

16bits du buffer.

Utilisé par initBuffer(), et shift().int Buffer::libre

Espace libre dans la partie donnée des 16bits du buffer.

Utilisé par initBuffer(), et shift().int Buffer::tamponUtilise

Espace utilise dans la partie tampon des 16bits du buffer.

Utilisé par initBuffer(), et shift().

La documentation pour cette structure a été générée à partir du fichier suivant :

- **Buffer.h**

b) CodeCaractere

```
#include <CodeCaractere.h>
```

Champs de données

1. char * **code**
2. int **longueurCode**

Description détaillée

Un **CodeCaractere** contient les informations sur une série de bits: Adresse de début et longueur.

Détaille des champs

char* CodeCaractere::code

Adresse du code sous forme de bits.

Utilisé par copieCode(), getCode(), nouveauCodeCaractere0(), setBit(), setCode(), setLongueurCodeMax(), et setNextBit().int CodeCaractere::longueurCode

Longueur en nombre de bits du code.

Utilisé par afficherCode(), afficherTableCodage(), copieCode(), getLongueurCode(), nouveauCodeCaractere0(), setBit(), setLongueurCode(), setLongueurCodeMax(), et setNextBit().

La documentation pour cette structure a été générée à partir du fichier suivant :

- **CodeCaractere.h**

c) ListeBits

```
#include <Buffer.h>
```

Champs de données

1. **ListeTmp liste**
2. int **intList**

Description détaillée

Union entre un int et **ListeTmp**. Permet d'effectuer des shift (<<) sur les 16 bits de **ListeTmp**. (Opération possible seulement sur les entiers).

La documentation pour cette union a été générée à partir du fichier suivant :

- **Buffer.h**

d) ListeTmp

```
#include <Buffer.h>
```

Champs de données

1. char **tampon**
2. char **code**

Description détaillée

Structure de taille 16 bits contenant deux caractères.

Détaille des champs

char ListeTmp::code

8 bits utilisés pour contenir un caractère qui peut être récupéré.

Utilisé par initBuffer().char ListeTmp::tampon

8 bits utilisés comme tampon du buffer.

Utilisé par initBuffer().

La documentation pour cette structure a été générée à partir du fichier suivant :

- **Buffer.h**

e) TableCodage

```
#include <TableCodage.h>
```

Champs de données

1. **CodeCaractere tableau** [NBCARACTERES+1]

Description détaillée

Tableau contenant à chaque indice i le code de huffman associé au caractère de code ascii i. Contient dans la dernière case le code associé au caractère spécial chargé d'identifier la fin de fichier.

Détaille des champs

CodeCaractere TableCodage::tableau[NBCARACTERES+1]

* Tableau de taille (nombre de caractères ascii +1).

Utilisé par afficherTableCodage(), calculeCode(), et parcoursArbre().

La documentation pour cette structure a été générée à partir du fichier suivant :

- **TableCodage.h**

f) TmpMaillon

```
#include <Maillon.h>
```

Champs de données

1. **TmpMaillon * suivant**
2. **Tas arbre**

Description détaillée

Maillon: Maillon d'une liste chaînée.

Détaille des champs

Tas TmpMaillon::arbre

Arbre de codage: Tas.

Voir aussi :

Tas

Utilisé par getArbre(), getPoid(), nouveauMaillon0(), nouveauMaillon1(), et setArbre().struct TmpMaillon* TmpMaillon::suivant

Pointeur sur le maillon suivant de la liste. Mis à NULL si fin de liste.

Utilisé par getSuivant(), nouveauMaillon0(), et setSuivant().

La documentation pour cette structure a été générée à partir du fichier suivant :

- **Maillon.h**

g) TmpNoeud

```
#include <Noeud.h>
```

Champs de données

1. char **donnee**
2. int **estEOF**:1
3. int **poidsCumules**
4. **TmpNoeud * filsDroit**
5. **TmpNoeud * filsGauche**

Description détaillée

Noeud: Noeud d'un Tas. Noeud: Noeud d'un Tas. Entête du module offrant une structure de Noeud. Contient l'adresse de ses fils droit et gauche s'il s'agit d'un nœud ainsi que le poids cumulé. Contient un caractère et son nombre d'occurrence s'il s'agit d'une feuille.

Détaille des champs

char TmpNoeud::donnee

Caractère: information située dans une feuille: le parcours du tas jusqu'à une feuille donne un code associé à ce caractère.

Utilisé par getDonnee(), nouveauNoeud1(), et setDonnee().int TmpNoeud::estEOF

Permet de savoir si la feuille est le caractère spécial utilisé pour identifier la fin de fichier: information située dans une feuille.

Utilisé par isEOF(), nouveauNoeud0(), et setEOF().int TmpNoeud::poidsCumules

Poids cumulé des noeuds situés en dessous du noeud courant (fils droit et fils gauche). Valeur du nombre d'occurrence du caractère donnée si on est sur une feuille.

Utilisé par getPoidsCumules(), nouveauNoeud1(), nouveauNoeud2(), et setPoidsCumules().

La documentation pour cette structure a été générée à partir du fichier suivant :

- **Noeud.h**

4.2.2 Description des modules et de leurs fonctions

a) Buffer.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Buffer.h"
```

Fonctions

1. IMPORT void **initBuffer** (**Buffer** *courant)
2. IMPORT int **shift** (**Buffer** *courant)

Description détaillée

Module offrant une structure de buffer d'une longueur de 16 bits. Utilisé pour l'écriture des codes caractères dans l'archive 8 bits par 8 bits.

Description des fonctions

IMPORT void initBuffer (Buffer * *courant*)

Initialisation d'un buffer. Les 16 bits sont initialisés à 0. Il y a 8 bits libres dans la partie gauche du buffer (partie utilisée pour écrire). Le tampon utilisé est de 0 bits.

Paramètres :

courant Adresse du buffer à initialiser.

Références ListeTmp::code, Buffer::codeTampon, Buffer::libre, ListeBits::liste, ListeTmp::tampon, et Buffer::tamponUtilise.IMPORT int shift (Buffer * *courant*)

Effectue un shift des données contenues dans la partie tampon vers la partie code. Ce shift complète les bits libres de la partie code avec les bits présents dans la partie tampon.

Paramètres :

courant adresse du buffer sur lequel on veux effectuer un shift.

Returns:

1 si la partie code est pleine, 0 sinon.

Références Buffer::codeTampon, ListeBits::intList, Buffer::libre, et Buffer::tamponUtilise.

b) Buffer.h

```
#include "InfosComunes.h"
```

Structures des données

1. struct **ListeTmp**
2. union **ListeBits**
3. struct **Buffer**

Fonctions

1. void **initBuffer** (**Buffer** *courant)
2. int **shift** (**Buffer** *courant)

Description détaillée

Entête du module offrant une structure de buffer.

---

c) CodeCaractere.c

```
#include <stdio.h>
#include <stdlib.h>
#include "CodeCaractere.h"
```

Fonctions

1. IMPORT **CodeCaractere * nouveauCodeCaractere0** ()
2. IMPORT void **setCode** (**CodeCaractere** *courant, char *code)
3. IMPORT void **setLongueurCode** (**CodeCaractere** *courant, int longueurCode)
4. IMPORT void **setLongueurCodeMax** (**CodeCaractere** *courant, int longueurCode)
5. IMPORT void **copieCode** (**CodeCaractere** *source, **CodeCaractere** *destination)
6. IMPORT char * **getCode** (**CodeCaractere** *courant)
7. IMPORT int **getLongueurCode** (**CodeCaractere** *courant)
8. IMPORT void **setNextBit** (**CodeCaractere** *courant, int nouveauBit)
9. IMPORT void **setBit** (**CodeCaractere** *courant, int position, int nouveauBit)
10. IMPORT void **afficherCode** (**CodeCaractere** *courant)

Description détaillée

Module offrant une structure contenant un code d'une longueur de bits donnée.

Description des fonctions

IMPORT void afficherCode (CodeCaractere * *courant*)

Affiche le contenus d'un caractère bit a bits sous forme de 0 et 1.

Références CodeCaractere::longueurCode.

Utilisé par afficherTableCodage().IMPORT void copieCode (CodeCaractere * *source*, CodeCaractere * *destination*)

Copie le code contenu dans source dans destination. Alloue la mémoire nécessaire pour stocker le code. Le stockage se fait par groupes de 1octet.

Paramètres :

source Adresse du **CodeCaractere** dont on veut copier le code.
destination Adresse du **CodeCaractere** ou l'on veut copier le code.

Références CodeCaractere::code, et CodeCaractere::longueurCode.

Utilisé par parcoursArbre().IMPORT char* getCode (CodeCaractere * *courant*)

Retourne l'adresse du code courant.

Paramètres :

courant Adresse du **CodeCaractere** dont on veut le code.

Returns:

Adresse du code contenu dans courant.

Références CodeCaractere::code.IMPORT int getLongueurCode (CodeCaractere * *courant*)

Retourne la longueur du code courant.

Paramètres :

courant Adresse du **CodeCaractere** dont on veut la longueur du code.

Returns:

Longueur du code contenu dans courant.

Références CodeCaractere::longueurCode.IMPORT CodeCaractere* nouveauCodeCaractere0 ()

Crée un nouveau **CodeCaractere**. Pointeur vers le code est initialise à NULL. La longueur est initialisée à 0.

Returns:

adresse du nouveau **CodeCaractere**.

Références CodeCaractere::code, et CodeCaractere::longueurCode.IMPORT void setBit (CodeCaractere * *courant*, int *position*, int *nouveauBit*)

Ajoute le bit nouveauBit à la position position dans le code déjà présent. L'accès à la mémoire se faisant octet par octet, on utilise un système de masque pour modifier le bit voulu.

Paramètres :

courant Adresse du **CodeCaractere** dans lequel on veut ajouter un bit au code.
position Position ou le bit doit être inséré.
nouveauBit Valeur (0 ou 1) du bit a insérer.

Références CodeCaractere::code, et CodeCaractere::longueurCode.

Utilisé par parcoursArbre().IMPORT void setCode (CodeCaractere * *courant*, char * *code*)

Définit l'adresse du code caractère du module courant. Si code n'est pas nulle le pointeur est mis a jours, sinon l'ancienne valeur est conservée.

Paramètres :

courant Adresse du **CodeCaractere** dont on veut mettre a jours le code.
code adresse du nouveau code.

Références CodeCaractere::code.IMPORT void setLongueurCode (CodeCaractere * *courant*, int *longueurCode*)

Définie la longueur du code.

Paramètres :

courant Adresse du **CodeCaractere** dont on veut mettre a jours la longueur du code.
longueurCode Nouvelle longueur du code.

Références CodeCaractere::longueurCode.

Utilisé par calculeCode().IMPORT void setLongueurCodeMax (CodeCaractere * *courant*, int *longueurCode*)

Définie la longueur maximale du code. Alloue la mémoire nécessaire pour stocker un code de longueurCode bits. Le stockage se fait par groupes de 1octet.

Paramètres :

courant Adresse du **CodeCaractere** dont on définir la longueur maximale.
longueurCode longueur maximale du code.

Références CodeCaractere::code, et CodeCaractere::longueurCode.

Utilisé par calculeCode().IMPORT void setNextBit (CodeCaractere * *courant*, int *nouveauBit*)

Ajoute le bit nouveauBit a la fin du code déjà présent. L'accès à la mémoire se fessant octet par octet, on utilise un système de masque pour modifier le bit voulu. La longueur du code est mise à jour. Fonction utilisée pendant la compression lors du parcours de l'arbre pour déterminer les codes de chaque caractère.

Paramètres :

courant Adresse du **CodeCaractere** dans lequel on veut ajouter un bit au code.
nouveauBit Valeur (0 ou 1) du bit a insérer a la fin du code.

Références CodeCaractere::code, et CodeCaractere::longueurCode.

d) CodeCaractere.h

```
#include "InfosComunes.h"
```

Structures des données

1. struct **CodeCaractere**

Fonctions

1. IMPORT **CodeCaractere** * **nouveauCodeCaractere0** ()
2. IMPORT void **setCode** (**CodeCaractere** *courant, char *code)
3. IMPORT void **setLongueurCode** (**CodeCaractere** *courant, int longueurCode)
4. IMPORT void **setLongueurCodeMax** (**CodeCaractere** *courant, int longueurCode)
5. IMPORT void **copieCode** (**CodeCaractere** *source, **CodeCaractere** *destination)
6. IMPORT char * **getCode** (**CodeCaractere** *courant)
7. IMPORT int **getLongueurCode** (**CodeCaractere** *courant)
8. IMPORT void **setBit** (**CodeCaractere** *courant, int position, int nouveauBit)
9. IMPORT void **afficherCode** (**CodeCaractere** *courant)

Description détaillée

Entête du module offrant une structure contenant un code d'une longueur de bits donnée.

---

e) Compression.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Compression.h"
```

Fonctions

1. PRIVATE int * **lectureFichier** (char nomFichierSource[])
2. PRIVATE **TableCodage * calculeCode** (**Tas** codes)
3. PRIVATE void **parcoursArbre** (**Tas** codes, **TableCodage** *listeCodes, **CodeCaractere** *codeCouran, int position)
4. PRIVATE void * **ecritureArchive** (FILE *archive, char nomFichierSource[], **TableCodage** *listeCodes, int *listeOccurences)
5. IMPORT void * **compression** (int nbFichiers, char *nomsFichiers[], char nomArchive[])
6. IMPORT **Tas constructionArbreCodage** (int *listeOccurences)
7. IMPORT void **afficheListe** (int *listeOccurences)
8. IMPORT void **afficheArbre** (**Tas** arbre)
9. IMPORT void **afficherListe** (**ListeTriee** liste)
10. IMPORT void **afficherTableCodage** (**TableCodage** *listeCodes)

Description détaillée

Module de gestion de la compression des fichiers.

Description des fonctions

IMPORT void afficheArbre (Tas *arbre*)

Fonction d'affichage d'un arbre de codage. Noeud : affichage du poids cumulé. Feuille : affichage du poids et du caractère correspondant. (Identification du caractère spécial EOF).

Paramètres :

arbre Arbre de codage.

Références estFeuille(), getDonnee(), getFilsDroit(), getFilsGauche(), getPoidsCumules(), et isEOF().

Utilisé par afficherListe().IMPORT void afficheListe (int * *listeOccurences*)

Fonction affichant les informations contenues dans une liste d'occurrences : code ASCII et nombre d'occurrence (si>0).

Paramètres :

listeOccurences Liste contenant à chaque indice i le nombre d'occurrence du caractère avec le code ASCII i.

IMPORT void afficherListe (ListeTriee *liste*)

Fonction d'affichage d'une liste chaîne triée d'arbres.

Paramètres :

liste Liste triée avec un arbre dans chaque maillon.

Références afficheArbre(), getArbre(), et getSuivant().IMPORT void afficherTableCodage (TableCodage * *listeCodes*)

Fonction d'affichage d'une table de codage. Pour chaque caractère avec un code de longueur supérieur a 0: Affichage du code ascii, de la longueur du code, du code associe à ce caractère.

Paramètres :

listeCodes Liste contenant à chaque indice i un **CodeCaractere** contenant une longueur et un code sous forme de bits, correspondant au caractère avec le code ASCII i.

Références afficherCode(), CodeCaractere::longueurCode, et TableCodage::tableau.PRIVATE TableCodage * calculeCode (Tas *codes*)

Calcule la table de codage à partir de l'arbre de codage.

Voir aussi :

void **parcoursArbre(Tas codes,TableCodage* listeCodes,CodeCaractere* codeCouran,int position)**

Paramètres :

codes Arbre de codage.

Returns:

Liste contenant à chaque indice i un **CodeCaractere** contenant une longueur et un code sous forme de bits, correspondant au caractère avec le code ASCII i. NULL en cas d'échec.

Références setLongueurCode(), setLongueurCodeMax(), et TableCodage::tableau.IMPORT void* compression (int *nbFichiers*, char * *nomsFichiers*[], char *nomArchive*[])

Fonction se chargeant de la compression d'une liste de fichiers dans une archive de nom donné. Apres la création de l'archive: Pour chaque fichiers: - calcule du nombre d'occurrences des caractères.

- calcule de l'arbre de codage.
- calcule des codes associes a chaque caractères. -écritures des données compressées dans l'archive. Fermeture de l'archive. Affichage de messages d'erreur en cas de problème pendant la compression.
- **Parameters :**

  nbFichiers Nombre de fichiers à compresser.
  nomsFichiers Liste des noms de fichiers à compresser.
  nomArchive Nom de l'archive ou doivent être compresses les fichiers.
- **Returns:**

  !NULL si succès, NULL en cas d'échec.

Références lectureFichier().

Utilisé par main().IMPORT Tas constructionArbreCodage (int * *listeOccurences*)

Fonction créant un arbre de codage à partir d'une liste d'occurrence. Cree une liste chaînée triée de façon décroissante suivant le nombre d'occurrences trouve dans la liste d'occurrences d'arbre à une feuille. Ajout d'un maillon avec une feuille de poids 0 qui sera utilisé affin d'avoir un code identifiant la fin des données compressées d'un fichier (EOF). Chaque maillon correspondant a une lettre de la liste d'occurrence avec une occurrence supérieur a 0. Fusionne deux a deux les maillons contenant les arbres avec les poids cumules les plus faibles. Termine quant il ne reste plus qu'un maillon contenant un arbre fusion de toutes les feuilles de départ. Fonction exportée car utilisée lors de la décompression.

Paramètres :

listeOccurences Liste contenant à chaque indice i le nombre d'occurrence du caractère avec le code ASCII i.

Returns:

Arbre de codage correspondant aux infos pressentes dans listeOccurences plus le caractère spécial de fin de fichier, NULL si échec.

Références ajouterMaillon(), creerListeVide(), nouveauMaillon1(), nouveauNoeud0(), nouveauNoeud1(), setEOF(), et setPoidsCumules().PRIVATE void * ecritureArchive (FILE * ***archive*****, char** ***nomFichierSource*****[], TableCodage *** ***listeCodes*****, int *** ***listeOccurences*****)**

Ecriture dans l'archive des données compressées correspondant au contenus du fichier source grâce à la table de codage. L'archive doit déjà être ouverte pour permettre l'ajout d'un nouveau fichier compressé dans une archive ou il y en a déjà. Ecrit le nom du fichier compressé, puis un saut de ligne. Ecrit la liste d'occurrence des caractère qui apparaissent au moins une fois puis un saut de ligne (termine la liste d'occurrence avec un caractère d'occurrence 0 pour identifier la fin de la liste). Utilise un buffer pour écrire caractères par caractères les données compressées : pour chaque caractères du fichier a compresser, écriture dans l'archive du codes associé dans la table de codage. Ajout à la fin du code du caractère identifiant la fin des données de ce fichier compressé.

Paramètres :

archive Fichier dans lequel les données compressées sont écrites.
nomFichierSource Nom du fichier à compresser.
listeCodes Liste des codes associés aux caractères du fichier à compresser.
listeOccurences Liste d'occurrences associée aux caractères du fichier à compresser.

Returns:

!NULL en cas de succès. NULL en cas d'échec.

PRIVATE int * lectureFichier (char ***nomFichierSource*****[])**

Parcours le fichier nomFichierSource et identifie le nombre d'occurrences de chaque caractère.

Paramètres :

nomFichierSource nom du fichier à analyser.

Returns:

Liste des occurrences de chaque caractère de la table ASCII. Tableau avec a l'indice i le nombre d'occurrences du caractère avec le code ascii i. NULL si le fichier source n'existe pas.

Utilisé par compression().PRIVATE void parcoursArbre (Tas ***codes*****, TableCodage *** ***listeCodes*****, CodeCaractere *** ***codeCouran*****, int** ***position*****)**

Pour chaque noueux : Si ce n'est pas une feuille :

- ajout d'un 0 au code courant, parcours ers la gauche.
- ajout d'un 1 au code courant, parcours ers la droite. Si c'est une feuille :
- copie du code courant dans la table de codage à l'indice correspondant au caractère de la feuille. (Traitement identique pour le code spécial EOF, placé en fin de la table de codage).

Paramètres :

codes Arbre de codage a sa position courante.
listeCodes table de codage où sont places les codes correspondant aux caractères trouves dans les feuilles de l'arbre.
codeCouran Contient le code correspondant au parcours de l'arbre effectues lors des précédents appels récursifs.
position Longueur du codeCouran.

Références copieCode(), estFeuille(), getDonnee(), getFilsDroit(), getFilsGauche(), isEOF(), setBit(), et TableCodage::tableau.

f) ListeTriee.c

```
#include <stdio.h>
#include <stdlib.h>
#include "ListeTriee.h"
```

Fonctions

1. IMPORT **ListeTriee creerListeVide** ()
2. IMPORT **ListeTriee ajouterMaillon** (**ListeTriee** listeCourante, **Maillon** *nouveauMaillon)
3. IMPORT **ListeTriee supprimerTete** (**ListeTriee** listeCourante)

Description détaillée

Module offrant une liste chaînée triée contenant des arbres. Le tris est effectué suivant la valeur du poids cumule a la racine de l'arbre. Les fonctions maintiennent la liste triée.

Description des fonctions

IMPORT ListeTriee ajouterMaillon (ListeTriee *listeCourante*, Maillon * *nouveauMaillon*)

Insert le nouveau maillon dans la liste. Si la liste est vide, retourne l'adresser du nouveau maillon. Sinon: parcours la liste pour trouver la position d'insertion qui respecte l'ordre décroissant. Si un ou plusieurs maillons on déjà le même poids, que le nouveau maillon, il est insère après ceux-ci, le plus loin possible dans la liste.

Paramètres :

listeCourante Adresse du premier élément de la liste.
nouveauMaillon Adresse du maillon à insérer dans la liste.

Returns:

Adresse du début de la liste triée.

Références getPoid(), getSuivant(), et setSuivant().

Utilisé par constructionArbreCodage().IMPORT ListeTriee creerListeVide ()

Une liste triée vide est un pointeur de valeur NULL.

Returns:

NULL: liste vide.

Utilisé par constructionArbreCodage().IMPORT ListeTriee supprimerTete (ListeTriee *listeCourante*)

Supprime l'élément en tête de liste.

Paramètres :

listeCourante Adresse du premier élément de la liste.

Returns:

Adresse du début de la liste triée après suppression de l'élément de tête. NULL si la liste est vide.

Références detruire(), et getSuivant().

g) ListeTriee.h

```
#include "Maillon.h"
```

Typedefs

1. typedef **Maillon * ListeTriee**

Fonctions

1. IMPORT **ListeTriee creerListeVide** ()
2. IMPORT **ListeTriee ajouterMaillon** (**ListeTriee** listeCourante, **Maillon** *nouveauMaillon)
3. IMPORT **ListeTriee supprimerTete** (**ListeTriee** listeCourante)

Description détaillée

Entête du module offrant une liste chaînée triée contenant des arbres.

Description des Typedef

typedef Maillon* ListeTriee

Une ListeTriee est en fait un pointeur sur un Maillon.

Voir aussi :

Maillon

---

h) Maillon.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Maillon.h"
```

Fonctions

1. IMPORT **Maillon * nouveauMaillon0** ()
2. IMPORT **Maillon * nouveauMaillon1** (**Tas** arbre)
3. IMPORT void **detruire** (**Maillon** *courant)
4. IMPORT int **getPoid** (**Maillon** *courant)
5. IMPORT **Maillon * getSuivant** (**Maillon** *courant)
6. IMPORT int **setSuivant** (**Maillon** *courant, **Maillon** *suivant)
7. IMPORT **Tas getArbre** (**Maillon** *courant)
8. IMPORT void **setArbre** (**Maillon** *courant, **Tas** arbre)

Description détaillée

Module offrant une Maillon utilise dans liste triée. Permet d'accéder au suivant et contient un arbre de type Tas.

Voir aussi :

ListeTriee

Description des fonctions

IMPORT void detruire (Maillon * *courant*)

Détruit le maillon courant. Note: Pas de désallocation de l'arbre.

Paramètres :

courant maillon à détruire.

Utilisé par supprimerTete().IMPORT Tas getArbre (Maillon * *courant*)

Permet d'accéder au Tas contenus dans le maillon.

Paramètres :

courant Maillon dont on veut le Tas.

Returns:

Racine de l'arbre contenu dans le maillon.

Références TmpMaillon::arbre.

Utilisé par afficherListe().IMPORT int getPoid (Maillon * *courant*)

Permet d'accéder au poids cumule situe en haut de l'arbre.

Paramètres :

courant maillon dont on veut le poids de l'arbre qu'il contient.

Returns:

valeur du poids cumule situe en haut de l'arbre.

Références TmpMaillon::arbre, et getPoidsCumules().

Utilisé par ajouterMaillon().IMPORT Maillon* getSuivant (Maillon * *courant*)

Permet d'accéder au maillon suivant.

Paramètres :

courant Maillon dont on veut le suivant.

Returns:

Adresse du maillon suivant. (NULL si courant est le dernier de la liste ou si courant=NULL).

Références TmpMaillon::suivant.

Utilisé par afficherListe(), ajouterMaillon(), et supprimerTete().IMPORT Maillon* nouveauMaillon0 ()

Crée un nouveau maillon. Allocation de la mémoire, arbre et suivant mis à NULL.

Returns:

Adresse du nouveau maillon.

Références TmpMaillon::arbre, et TmpMaillon::suivant.

Utilisé par nouveauMaillon1().IMPORT Maillon* nouveauMaillon1 (Tas *arbre*)

Crée un nouveau maillon. Allocation de la mémoire, arbre suivant mis à NULL, arbre pointe sur l'arbre donne en paramètre.

Paramètres :

arbre à insérer dans le maillon.

Returns:

Adresse du nouveau maillon.

Références TmpMaillon::arbre, et nouveauMaillon0().

Utilisé par constructionArbreCodage().IMPORT void setArbre (Maillon * *courant*, Tas *arbre*)

Permet de définir l'arbre contenu dans le maillon courant.

Paramètres :

courant Maillon dont on veut définir l'arbre.
arbre Tas que l'on veut insérer dans le maillon.

Références TmpMaillon::arbre.IMPORT int setSuivant (Maillon * *courant*, Maillon * *suivant*)

Permet de définir le maillon suivant du maillon courant.

Paramètres :

courant Maillon dont on veut définir le suivant.
suivant Maillon que l'on veut mettre comme suivant.

Returns:

0 si succès, -1 si courant=NULL ou suivant=NULL.

Références TmpMaillon::suivant.

Utilisé par ajouterMaillon().

i) Maillon.h

```
#include "Tas.h"
```

Structures des données

1. struct **TmpMaillon**

Typedefs

1. typedef **TmpMaillon Maillon**

Fonctions

1. IMPORT **Maillon** * **nouveauMaillon0** ()
2. IMPORT **Maillon** * **nouveauMaillon1** (**Tas** arbre)
3. IMPORT void **detruire** (**Maillon** *courant)
4. IMPORT int **getPoid** (**Maillon** *courant)
5. IMPORT **Maillon** * **getSuivant** (**Maillon** *courant)
6. IMPORT int **setSuivant** (**Maillon** *courant, **Maillon** *suivant)
7. IMPORT **Noeud** * **getArbre** (**Maillon** *courant)
8. IMPORT void **setArbre** (**Maillon** *courant, **Tas** noeud)

Description détaillée

Entête du module offrant une Maillon utilise dans ListeTriee.

Description des Typedef

typedef struct TmpMaillon Maillon

Maillon: Maillon d'une liste chaînée.

---

j) Noeud.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Noeud.h"
```

Fonctions

1. IMPORT **Noeud * nouveauNoeud0** ()
   Cree un nouveau Noeud et retourne son adresse.

2. IMPORT **Noeud * nouveauNoeud1** (char donnee, int poidsCumules)
   Cree un nouveau Noeud et retourne son adresse, ce noeud est une feuille.

3. IMPORT **Noeud * nouveauNoeud2** (int poidsCumules, **Noeud** *filsDroit, **Noeud** *filsGauche)
   Cree un nouveau Noeud et retourne son adresse, ce noeud est un noeud, non une feuille.

4. IMPORT int **estFeuille** (**Noeud** *nCourant)
   Indique si le noeud est une feuille.

5. IMPORT char **getDonnee** (**Noeud** *nCourant)
   Retourne le caractère contenu dans le noeud.

6. IMPORT void **setDonnee** (**Noeud** *nCourant, char donnee)
   Affecte au caractère contenu dans la noeud la valeur de donnee.

7. IMPORT int **getPoidsCumules** (**Noeud** *nCourant)
   Retourne la valeur de poidsCumules contenue dans le noeud.

8. IMPORT void **setPoidsCumules** (**Noeud** *nCourant, int poidsCumules)
   Affecte au poids contenu dans la noeud la valeur de poidsCumules.

9. IMPORT **Noeud * getFilsDroit** (**Noeud** *nCourant)
   Retourne l'adresse du fils droit du noeuds.

10. IMPORT void **setFilsDroit** (**Noeud** *nCourant, **Noeud** *filsDroit)
    Affecte un noeuds comme fils droit au noeud nCourant.

11. IMPORT **Noeud * getFilsGauche** (**Noeud** *nCourant)
    Retourne l'adresse du fils gauche du noeuds.

12. IMPORT void **setFilsGauche** (**Noeud** *nCourant, **Noeud** *filsGauche)
    Affecte un noeuds comme fils gauche au noeud nCourant.

13. IMPORT int **isEOF** (**Noeud** *nCourant)
    Indique si le noeud contient le caractère spécial EOF.

14. IMPORT void **setEOF** (**Noeud** *nCourant)

Positionne la valeur estEOF de nCourant a 1.

Description détaillée

Module offrant une structure de Noeud utilise pour Tas. Noeud peut être une feuille: filsDroit=NULL et filsGauche=NULL. Noeud peut être un noeud de l'arbre: filsDroit!=NULL et filsGauche!=NULL. Ces propriétés sont garanties par les fonctions du module.

Documentation des fonctions

IMPORT int estFeuille (Noeud * *nCourant*)

Indique si le noeud est une feuille.

Paramètres:

nCourant Noeud dont on cherche à savoir si il s'agit d'une feuille.

Renvoie:

1 si il s'agit d'une feuille, 0 sinon.

IMPORT char getDonnee (Noeud * *nCourant*)

Retourne le caractère contenu dans le noeud.

Paramètres:

nCourant Noeud dont on cherche le caractère.

Renvoie:

caractères contenus dans le noeud.

IMPORT Noeud* getFilsDroit (Noeud * *nCourant*)

Retourne l'adresse du fils droit du noeuds.

Paramètres:

nCourant Noeud dont on cherche le fils droit.

Renvoie:

adresse du fils droit de nCourant.

IMPORT Noeud* getFilsGauche (Noeud * *nCourant*)

Retourne l'adresse du fils gauche du noeuds.

Paramètres:

nCourant Noeud dont on cherche le fils gauche.

Renvoie:

adresse du fils gauche de nCourant.

IMPORT int getPoidsCumules (Noeud * *nCourant*)

Retourne la valeur de poidsCumules contenue dans le noeud.

Paramètres:

nCourant Noeud dont on cherche le poidsCumules.

Renvoie:

valeur de poidsCumules dans nCourant.

IMPORT int isEOF (Noeud * *nCourant*)

Indique si le noeud contient le caractère spécial EOF.

Paramètres:

nCourant Noeud dont on veut tester estEOF.

Renvoie:

1 si le noeud est le caractère EOF, 0 sinon.

IMPORT Noeud* nouveauNoeud0 ()

Cree un nouveau Noeud et retourne son adresse.

Les valeurs sont initialisés à 0 ou NULL.

Renvoie:

L'adresse du nouveau noeud.

IMPORT Noeud* nouveauNoeud1 (char *donnee*, int *poidsCumules*)

Cree un nouveau Noeud et retourne son adresse, ce noeud est une feuille.

Les valeurs des fils sont initialisés à NULL, estEOF a 0. Les valeurs de donnée et poidsCumules sont initialisées avec les paramètres.

Paramètres:

donnee caractère que l'on souhaite stocker dans la feuille.
poidsCumules valeur associée au caractère.

Renvoie:

L'adresse du nouveau noeud.

IMPORT Noeud* nouveauNoeud2 (int *poidsCumules*, Noeud * *filsDroit*, Noeud * *filsGauche*)

Cree un nouveau Noeud et retourne son adresse, ce noeud est un noeud, non une feuille.

La valeur de estEOF et donnée sont initialisées a 0. Les valeurs des fils et du poids cumule sont initialisées avec les paramètres.

Paramètres:

poidsCumules valeur correspondant au poids cumulés de tous les fils.
filsDroit Adresse du fils droit.
filsGauche Adresse du fils gauche.

Renvoie:

L'adresse du nouveau noeud.

IMPORT void setDonnee (Noeud * *nCourant*, char *donnee*)

Affecte au caractère contenu dans la noeud la valeur de donnee.

Paramètres:

nCourant Noeud dont on veut changer la valeur du caractère.
donnee Nouvelle valeur du caractère contenus dans le noeud.

IMPORT void setEOF (Noeud * *nCourant*)

Positionne la valeur estEOF de nCourant a 1.

Notez que cette valeur ne peu pas être remise a 0.

Paramètres:

nCourant Noeud dont on veut mettre estEOF a 1.

IMPORT void setFilsDroit (Noeud * *nCourant*, Noeud * *filsDroit*)

Affecte un noeuds comme fils droit au noeud nCourant.

Paramètres:

nCourant Noeud dont on veut changer le fils droit.
filsDroit Adresse du nouveau fils droit.

IMPORT void setFilsGauche (Noeud * *nCourant*, Noeud * *filsGauche*)

Affecte un noeuds comme fils gauche au noeud nCourant.

Paramètres:

nCourant Noeud dont on veut changer le de fils gauche.
filsGauche Adresse du nouveau fils gauche.

IMPORT void setPoidsCumules (Noeud * *nCourant*, int *poidsCumules*)

Affecte au poids contenu dans la noeud la valeur de poidsCumules.

Paramètres:

nCourant Noeud dont on veut changer le poids.
poidsCumules Nouvelle valeur du poids.

k) Noeud.h

```
#include "InfosComunes.h"
```

Structures de données

1. struct **TmpNoeud**
   Noeud: Noeud d'un Tas.

Typedefs

1. typedef **TmpNoeud Noeud**
   Noeud: Noeud d'un Tas.

Fonctions

1. IMPORT **Noeud * nouveauNoeud0** ()
   Cree un nouveau Noeud et retourne son adresse.

2. IMPORT **Noeud * nouveauNoeud1** (char donnee, int poidsCumules)
   Cree un nouveau Noeud et retourne son adresse, ce noeud est une feuille.

3. IMPORT **Noeud * nouveauNoeud2** (int poidsCumules, **Noeud** *filsDroit, **Noeud** *filsGauche)
   Cree un nouveau Noeud et retourne son adresse, ce noeud est un noeud, non une feuille.

4. IMPORT int **estFeuille** (**Noeud** *nCourant)
   Indique si le noeud est une feuille.

5. IMPORT char **getDonnee** (**Noeud** *n)
   Retourne le caractère contenu dans le noeud.

6. IMPORT void **setDonnee** (**Noeud** *nCourant, char donnee)
   Affecte au caractère contenu dans la noeud la valeur de donnee.

7. IMPORT int **getPoidsCumules** (**Noeud** *nCourant)
   Retourne la valeur de poidsCumules contenue dans le noeud.

8. IMPORT void **setPoidsCumules** (**Noeud** *nCourant, int poidsCumules)
   Affecte au poids contenu dans la noeud la valeur de poidsCumules.

9. IMPORT **Noeud * getFilsDroit** (**Noeud** *nCourant)
   Retourne l'adresse du fils droit du noeuds.

10. IMPORT void **setFilsDroit** (**Noeud** *nCourant, **Noeud** *filsDroit)
    Affecte un noeuds comme fils droit au noeud nCourant.
11. IMPORT **Noeud * getFilsGauche** (**Noeud** *nCourant)
    Retourne l'adresse du fils gauche du noeuds.

12. IMPORT void **setFilsGauche** (**Noeud** *nCourant, **Noeud** *filsGauche)

Affecte un noeuds comme fils gauche au noeud nCourant.

13. IMPORT int **isEOF** (**Noeud** *nCourant)
    Indique si le noeud contient le caractère spécial EOF.

14. IMPORT void **setEOF** (**Noeud** *nCourant)
    Positionne la valeur estEOF de nCourant a 1.

Description détaillée

Entête du module offrant une structure de Noeud.

Description des Typedef

typedef struct TmpNoeud Noeud

Noeud: Noeud d'un Tas.

Noeud: Noeud d'un Tas. Entête du module offrant une structure de Noeud. Contient l'adresse des ses fils droit et gauche si il s'agit d'un nœud ainsi que le poids cumulé. Contient un caractère et son nombre d'occurrence si il s'agit d'une feuille.

---

l) TableCodage.c

```
#include "TableCodage.h"
```

Description détaillée

Module offrant une structure contenant les codes associés à chaque caractère de la table ASCII par Huffman.

---

m) TableCodage.h

```
#include "CodeCaractere.h"
#include "InfosComunes.h"
```

Structures des données

1. struct **TableCodage**

Description détaillée

Entête du module offrant une structure contenant les codes associes à chaque caractère de la table ASCII par Huffman.

---

n) Tas.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Tas.h"
```

Fonctions

1. IMPORT **Tas regrouper** (**Tas** tas1, **Tas** tas2)

Description détaillée

Module offrant une structure de tas: arbre binaire dont les noeuds sont ordonnés par leurs clefs respectives.

Permet de fusionner deux tas.

Voir aussi :

Noeud

Description des fonctions

IMPORT Tas regrouper (Tas *tas1*, Tas *tas2*)

Fusionne deux Tas. Cree un nouveau noeud avec les deux Tas pour fils droit et gauche et pour poids la somme des poids des deux Tas.

Returns:

Un tas fusion des deux tas.

Références getPoidsCumules(), et nouveauNoeud2().

o) Tas.h

```
#include "Noeud.h"
```

Typedefs

1. typedef **Noeud * Tas**

Fonctions

1. IMPORT **Tas regrouper** (**Tas** tas1, **Tas** tas2)

Description détaillée

Entête du module offrant une structure de tas.

Voir aussi :

Noeud

Description des Typedef

typedef Noeud* Tas

Un Tas est en fait un pointeur sur un Noeud.

Voir aussi :

Noeud

4.3 Module Decompression

a) Decompression.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Decompression.h"
```

Fonctions

1. PRIVATE char * **letureNomFichier** (FILE *archive)
   Lecture d'une chaîne dans le fichier archive.

2. PRIVATE int * **lectureTableCodage** (FILE *archive)
   Lecture de la table de codage dans le fichier.

3. PRIVATE void * **decodageFichier** (FILE *archive, char *nomFichier, **Tas** arbreCodage)
   Effectue la décompression des informations contenues dans l'archive pour le fichier nomFichier auquel est associé l'arbre de codage arbreCodage.

4. IMPORT void * **decompression** (char nomArchive[])
   Fonction se chargeant de la décompression d'une archive pouvant contenir plusieurs fichiers compressés.

5. IMPORT void **printBits** (char cha)
   Fonction d'affichage d'un octet bit à bit sous forme de 0 et de 1.

Description détaillée

Module de gestion de la décompression des fichiers.

Description des fonctions

PRIVATE void * decodageFichier (FILE * *archive*, char * *nomFichier*, Tas *arbreCodage*)

Effectue la décompression des informations contenues dans l'archive pour le fichier nomFichier auquel est associé l'arbre de codage arbreCodage.

archive doit être positionné sur le début des données compressées.

- Création du fichier destination.
- Lecture de l'archive octets par octets.
- pour chaque bit des octets lus:
  - parcours de l'arbre (0: à gauche, 1 à droite)
    - si on arrive sur une feuille: écriture du caractère associé.
  -

- Fermeture du fichier décompressé quand on trouve le code associé à la fin de fichier. Si on arrive à la fin de l'archive: l'archive est corrompue: on dois toujours finir la décompression en trouvant le code associé a la fin de fichier.

Paramètres:

archive Fichier déjà ouvert sur lequel on vas lire les données compressées: doit être positionné sur le début des données compressées.
nomFichier Adresse du nom du fichier qui doit être décompressé.
arbreCodage Arbre de codage associé au fichier à décompresser.

Renvoie:

!NULL si succès, NULL en cas d'échec.

IMPORT void* decompression (char *nomArchive*[])

Fonction se chargeant de la décompression d'une archive pouvant contenir plusieurs fichiers compressés.

Après l'ouverture de l'archive:

- Vérification du MagicNumber pour tester qu'il s'agit bien d'une archive jszip.
- Pour chaque fichiers contenus dans l'archive.
- Lecture du nom du fichier compressé.
- Lecture de la table de codage.
- Construction de l'arbre de codage.
- Lecture des données compressées dans l'archive et écriture des données décompressées dans un nouveau fichier.
- Fermeture de l'archive. Affichage de messages d'erreur en cas de problème pendant la décompression: pas une archive, archive corrompue.

Paramètres:

nomArchive Nom de l'archive a décompresser.

Renvoie:

!NULL si succès, NULL en cas d'échec.

PRIVATE int * lectureTableCodage (FILE * *archive*)

Lecture de la table de codage dans le fichier.

Allocation de l'espace mémoire pour contenir les informations lues. La lecture des occurrences se termine quand on rencontre un caractère avec une occurrence de 0: fin de la liste d'occurrence.

Paramètres:

archive Fichier déjà ouvert sur lequel on va lire les occurrences.

Renvoie:

Adresse d'un tableau de taille NBCARACTERES ou pour chaque indices i on trouve le nombre d'occurrences dans le fichier à décompresser du caractère avec le code ascii i.

PRIVATE char * letureNomFichier (FILE * *archive*)

Lecture d'une chaîne dans le fichier archive.

Allocation de l'espace mémoire pour contenir cette chaîne. La chaîne lue ne doit pas dépasser 512 caractères. Le caractère de saut de ligne en fin de chaîne est supprimé.

Paramètres:

archive Fichier déjà ouvert sur lequel la chaîne doit être lue.

Renvoie:

Adresse de la chaîne lue dans le fichier.

IMPORT void printBits (char *cha*)

Fonction d'affichage d'un octet bit à bit sous forme de 0 et de 1.

Paramètres:

cha Caractère à afficher bit par bit.

b) Decompression.h

```
#include "Compression.h"
#include "InfosComunes.h"
```

Fonctions

1. IMPORT void * **decompression** (char nomArchive[])
   Fonction se chargeant de la décompression d'une archive pouvant contenir plusieurs fichiers compressés.

2. IMPORT void **printBits** (char c)
   Fonction d'affichage d'un octet bit à bit sous forme de 0 et de 1.

Description détaillée

Entête du module de gestion de la décompression des fichiers.

Description des fonctions

IMPORT void* decompression (char *nomArchive*[])

Fonction se chargeant de la décompression d'une archive pouvant contenir plusieurs fichiers compressés.

Après l'ouverture de l'archive:

- Vérification du MagicNumber pour tester qu'il s'agit bien d'une archive jszip.
- Pour chaque fichiers contenus dans l'archive.
- Lecture du nom du fichier compressé.
- Lecture de la table de codage.
- Construction de l'arbre de codage.
- Lecture des données compressées dans l'archive et écriture des données décompressées dans un nouveau fichier.
- Fermeture de l'archive. Affichage de messages d'erreur en cas de problème pendant la décompression: pas une archive, archive corrompue.

Paramètres:

nomArchive Nom de l'archive a décompresser.

Renvoie:

!NULL si succès, NULL en cas d'échec.

IMPORT void printBits (char *cha*)

Fonction d'affichage d'un octet bit à bit sous forme de 0 et de 1.

Paramètres:

cha Caractère à afficher bit par bit.

4.4 Module Aide

Aide.h

```
#include "InfosComunes.h"
```

Fonctions

1. IMPORT void **affichageAide** (void)
   Fonction d'affichage de l'aide.

Description détaillée

Module d'affichage de l'aide.

Description des fonctions

IMPORT void affichageAide (void)

Fonction d'affichage de l'aide.

Affiche les informations sur le programme et son fonctionnement. !NULL si succès, NULL en cas d'échec.

5. Pseudos-Codes

5.1 Compression d'un fichier

```
Si le fichier existe :
     Initialisation de la table d'occurrence à 0 pour chaque
     caractère.
     Parcours du contenu caractère par caractère.
     Pour chaque caractères :
          Incrémenter le nombre d'occurrences de ce caractère.

     Construction de l'arbre de codage :
          Création d'une liste chaînée avec un arbre dans chaque
          maillon : arbre est une feuille avec un caractère et son
          poids. La liste est triée par ordre décroissant de poids.

          Jusqu'à ce qu'il reste un seul maillon :
               Fusion des arbres deux à deux et réinsertion dans la
               liste en conservant le tris.

     Calcule des codes Huffman :
          Parcours de l'arbre récursivement en profondeur :
          On mémorise : 0 quand on va à gauche, 1 quand on va à
          droite.
          Quand on arrive sur une feuille :
               Mémorisation du parcours effectué dans l'arbre pour
               le caractère contenus dans la feuille.

     Compression du fichier :
          Parcours du fichier source caractère par caractère.
          Pour chaque caractère :
               Ecriture du code associé dans le fichier.

          Ajout d'un code supplémentaire permettant d'identifier la
          fin des données compressées.
```

5.2 Décompression d'un fichier

```
Si l'archive existe :
     Lecture du nom du fichier compressé.

     Lecture de la table d'occurrence des caractères compressés.
          Tant qu'on ne trouve pas un poids de 0 :
               Lecture du code d'un caractère et de son poids.

     Construction de l'arbre de codage :
          Procédé identique à celui utilisé pendant la compression.

     Tant qu'on ne trouve pas le code de fin de fichier :
          Lecture octet par octet de l'archive :
          Pour chaque bit lus :
               Parcours de l'arbre de codage : 0 : vers la gauche,
               1 : vers la droite.
               Si on arrive sur une feuille :
                    Si la feuille contient un caractère :
                         Ecrire dans le fichier décompressé le
                         caractère trouvé dans la feuille.
                         Revenir à la racine de l'arbre.
                    Si la feuille identifie la fin de fichier :
                         Fermer le fichier.
```