Codage de Huffman

Le codage de Huffman (1952) est un codage statistique utilisé pour la compression sans perte de données telles que les textes, les images (fichiers JPEG) ou les sons (fichiers MP3). Dans le cas de textes, son principe est de définir un nouveau codage des caractères, codage à taille variable qui tient compte de la fréquence (le nombre d'occurrences) des caractères dans le texte : les caractères dont la fréquence est élevée seront codés sur moins de bits et ceux dont la fréquence est faible sur plus de bits.

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

1 Principe du codage de Huffman

Montrons le principe du codage de Huffman sur un exemple concret, un fichier qui contient le texte de la figure 1.

```
exemple de texte :
exempte tempete lexeme
```

FIGURE 1 – Le texte à compresser

La table 2 donne un codage obtenu par l'algorithme d'Huffman. Le caractère `'d'` a la fréquence la plus faible. Il apparaît 1 fois. Il est codé par `10101`, sur 5 bits. Le caractère avec la fréquence la plus élevée est `'e'` qui apparaît 15 fois et est codé par `11`, sur 2 bits.

Caractère	`\n`	␣	:	d	e	l	m	p	t	x
Code	0100	011	10100	10101	11	0101	000	1011	100	001

FIGURE 2 – Codage de Huffman

En appliquant ce codage sur le texte précédent (figure 1), on obtient l'encodage présenté à la figure 3. Les points sont utilisés pour séparer les groupes de bits correspondant aux codes des différents caractères. Le texte compressé tient sur 122 bits, soit 16 octets. L'original, codé en ASCII, utilise un octet par caractère, soit 42 octets et donc 336 bits.

```
11.001.11.000.1011.0101.11.011.10101.11.011.100.11.001.100.11.011.10100.⌟
↪ 0100.11.001.11.000.1011.100.11.011.100.11.000.1011.11.100.11.011.0101.⌟
↪ 11.001.11.000.11.0100
```

FIGURE 3 – Codage du texte d'origine en utilisant le code de la figure 2

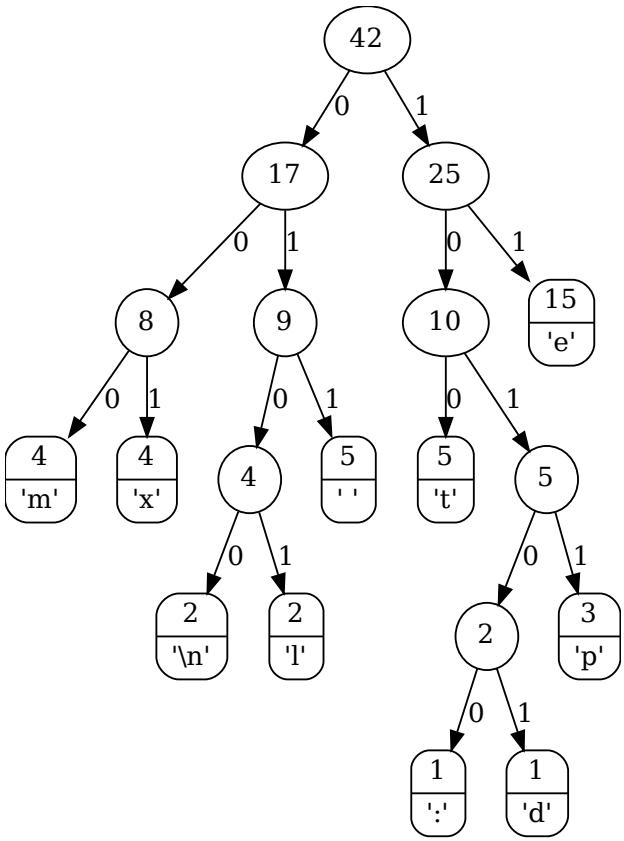

FIGURE 4 – Un arbre de Huffman avec les étiquettes sur les branches

L'algorithme de Huffman permet de définir un code optimal au sens de la plus courte longueur pour un codage par symbole. Il consiste à créer un arbre, dit *arbre de Huffman* (figure 4). Il s'agit d'un arbre binaire. Ses feuilles correspondent aux caractères du texte avec leur fréquence. Les nœuds ont toujours deux fils, un fils gauche et un fils droit : l'arbre binaire est dit parfait. Les nœuds permettent d'organiser les feuilles de manière à ce que la feuille dont le caractère a une fréquence élevée soit proche de la racine. À chaque nœud est associé une valeur qui est la somme de la valeur de son fils gauche et de la valeur de son fils droit. La valeur d'une feuille est la fréquence du caractère qu'elle représente. Ainsi, la valeur de la racine correspond au nombre de caractères du texte.

Pour définir le code de chaque caractère à partir de l'arbre de Huffman, le principe est le suivant. Chaque branche de l'arbre est étiquetée par la valeur 0 pour le sous-arbre gauche et la valeur 1 pour le sous-arbre droit (figure 4). Le code d'un caractère est alors la concaténation des étiquettes rencontrées (0 ou 1) en partant de la racine et en descendant jusqu'à la feuille contenant ce caractère. On retrouve bien les codes donnés figure 2.

Reste à construire cet arbre. La première étape consiste à calculer la fréquence des caractères du texte (figure 5). À partir de ces fréquences, on créé une liste d'arbres réduits à une feuille : chaque feuille correspond à un caractère du texte et à sa fréquence (figure 6). Ensuite, on crée un nouvel arbre à partir des deux arbres de fréquences les plus faibles qui en deviennent respec-

Caractère	\n	␣	:	d	e	l	m	p	t	x
Fréquence	2	5	1	1	15	2	4	3	5	4

FIGURE 5 – Fréquence des caractères du texte de la figure 1

tivement le sous-arbre gauche et le sous-arbre droit. Le sous-arbre de gauche est celui dont la fréquence est la plus faible. Ce nouvel arbre a pour fréquence la somme des fréquences de ses deux sous-arbres (figure 7). On recommence jusqu'à n'avoir qu'un seul arbre, l'arbre de Huffman (figures 8 à 15).

Notons que l'arbre de Huffman n'est pas unique. Il dépend de l'ordre des feuilles dans la liste initiale. En particulier, deux caractères de même fréquence pourraient être permutés.

2 Mise en œuvre

Nous avons vu le principe du codage de Huffman et sa construction. Il y a cependant des aspects pratiques et concrets à prendre en compte pour sa mise en œuvre. Nous les abordons dans les paragraphes suivants.

2.1 Fichier texte ou binaire

Un fichier dit texte est un fichier que l'on peut visualiser et modifier en utilisant un éditeur de texte. Cependant, parler de fichier texte n'est pas suffisant. Il faut connaître le codage utilisé pour être capable d'interpréter correctement son contenu. En effet, dans un fichier les données sont toujours conservées sous forme d'octets (suite de 8 bits : 0 ou 1). C'est le codage qui associe à ces suites de 0 et 1 les caractères correspondants. ASCII, Latin1, UTF-8... sont des exemples de codages. Par exemple, en ASCII le caractère « é » n'existe pas. En Latin1, il est représenté sur un octet (8 bits) donc la valeur est 233 en base 10 (E9 en hexadécimal ou 11101001 en binaire). En UTF-8, ce même caractère « é » est représenté sur deux octets (16 bits). Sa valeur est toujours 233 (compatibilité entre Latin1 et UTF-8) mais avec un premier octet dont tous les bits sont à 0 : 00E9 en hexadécimal ou 0000000011101001 en binaire.

L'encodage n'est pas une information transportée dans le fichier. Il peut y avoir des règles ou des conventions comme la balise META en HTML ou la mention de l'encodage en début des fichiers Python mais rien de systématique ni fiable. En conséquence, nous considèrerons tous les fichiers lus (et écrits) par nos programmes de compression et décompression comme des fichiers binaires. On lira donc des octets (8 bits correspondant à un entier compris entre 0 et 255), on calculera la fréquence de ces octets, etc.

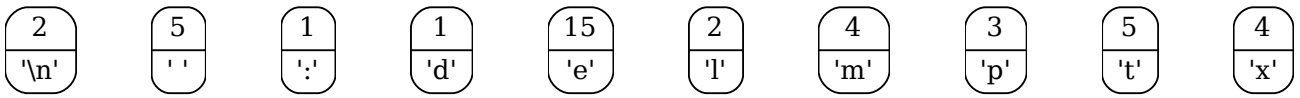

FIGURE 6 – Listes des arbres à l'étape 0

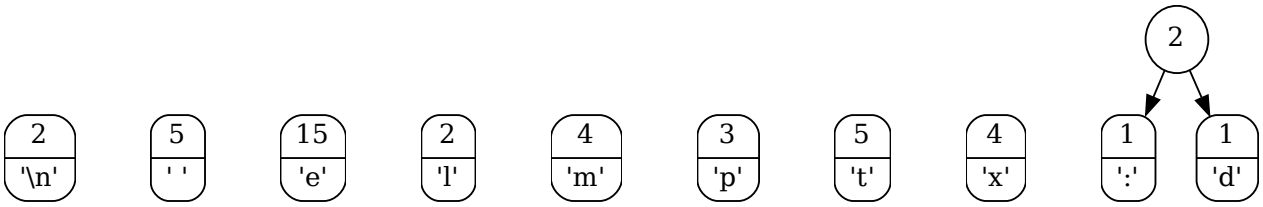

FIGURE 7 – Listes des arbres à l'étape 1

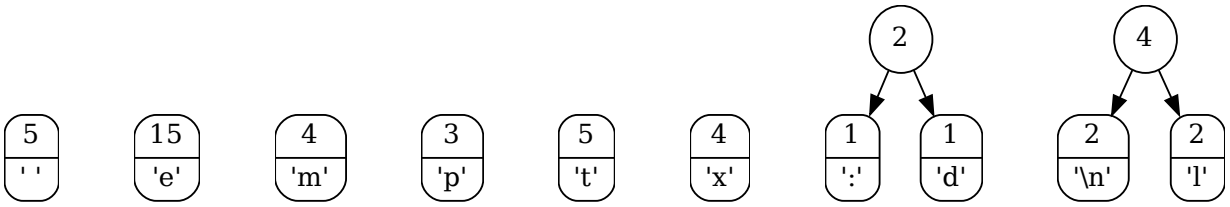

FIGURE 8 – Listes des arbres à l'étape 2

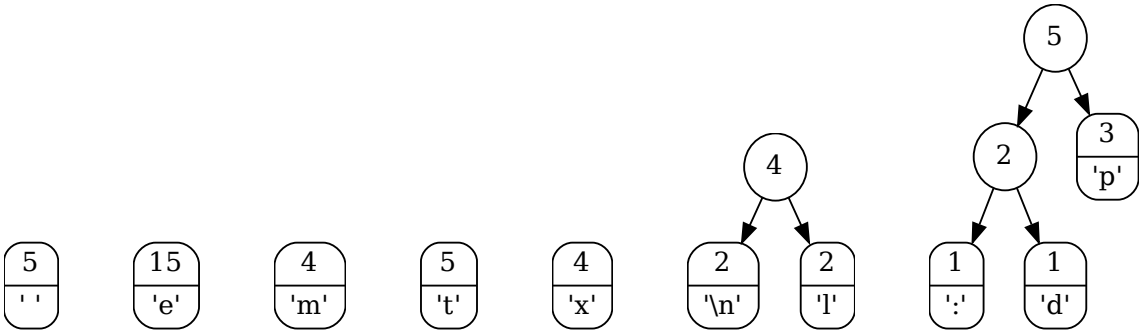

FIGURE 9 – Listes des arbres à l'étape 3

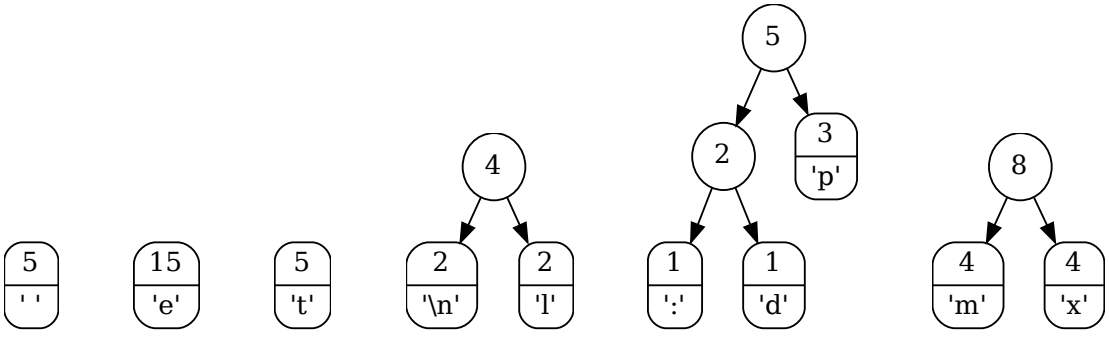

FIGURE 10 – Listes des arbres à l'étape 4

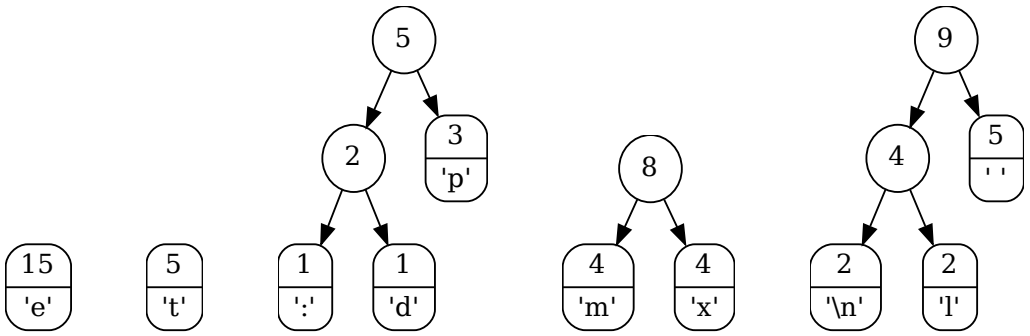

FIGURE 11 – Listes des arbres à l'étape 5

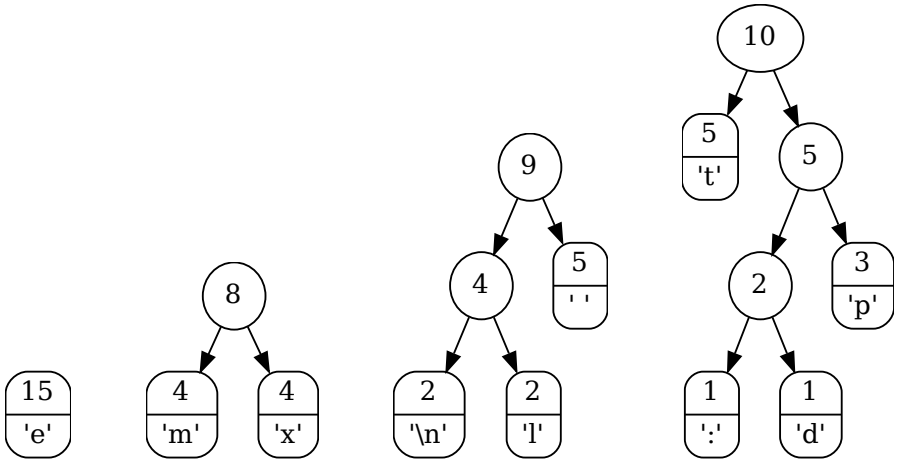

FIGURE 12 – Listes des arbres à l'étape 6

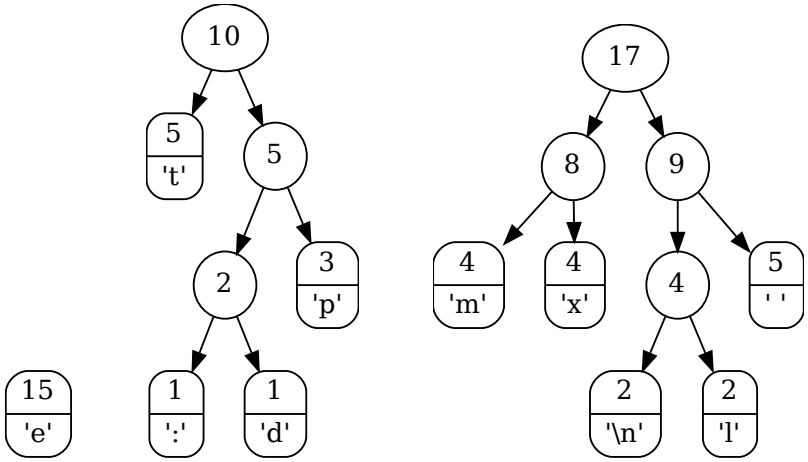

FIGURE 13 – Listes des arbres à l'étape 7

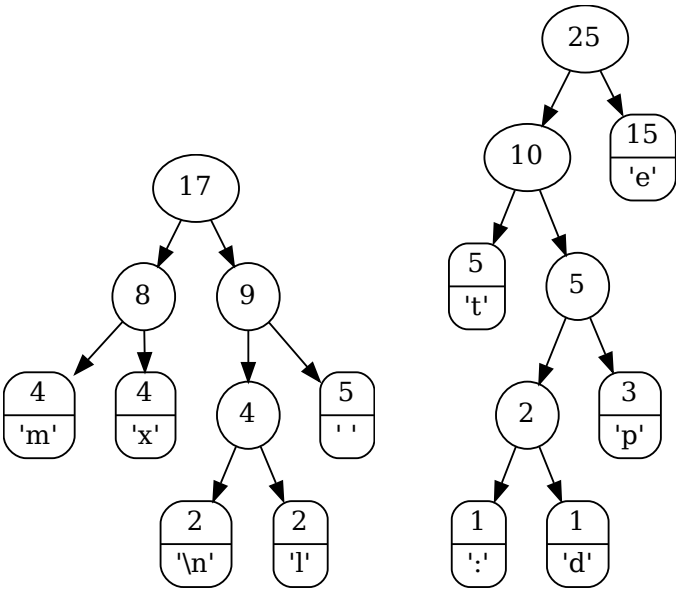

FIGURE 14 – Listes des arbres à l'étape 8

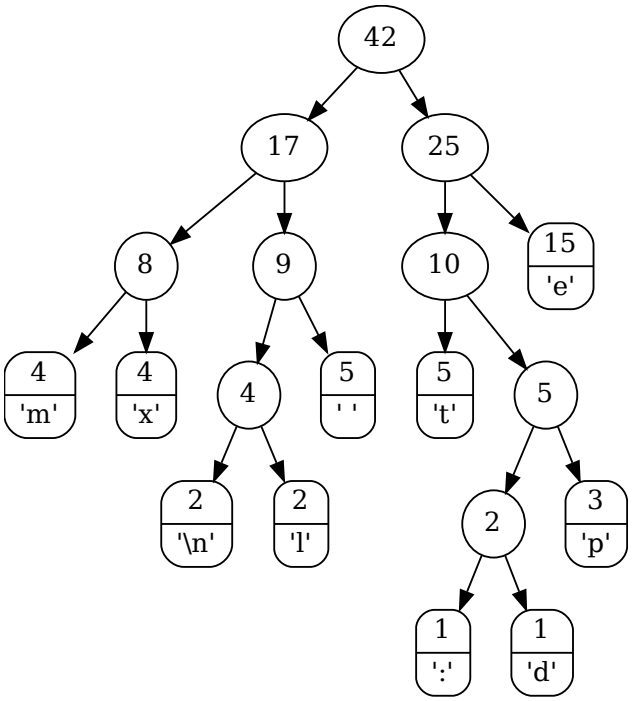

FIGURE 15 – Listes des arbres à l'étape 9

2.2 La taille du fichier

Le fichier compressé est un fichier dans lequel seront écrits des octets. Ainsi, les bits correspondant au codage du texte devront être regroupés par 8 pour former des octets (figure 16). Le dernier octet peut être incomplet. Dans le fichier, ce sont bien 8 bits qui seront écrits, les bits manquants étant certainement à 0. Lors de la décompression il ne faut pas interpréter ces bits surnuméraires. Il faut donc pouvoir déterminer quand la décompression est terminée.

```
11001110.00101101.01110111.01011101.11001100.11001101.11010001.00110011.⌋
↪  10001011.10011011.10011000.10111110.01101101.01110011.10001101.00
```

FIGURE 16 – Les bits de la figure 3 regroupés en octets

Une solution serait d'écrire au début du fichier compressé le nombre d'octets du texte d'origine. Se pose alors la question de comment stocker cet entier qui pourrait être très grand (et donc utiliser plusieurs octets).

Nous allons adopter ici une autre solution. Elle consiste à ajouter un nouveau symbole (en plus des 256 correspondant aux valeurs possibles pour un octet) qui marquera la fin du fichier. Lors de la décompression, quand ce symbole sera reconnu, on saura que le travail est terminé.

Bien sûr, il faut pouvoir représenter ce nouveau symbole. Sa valeur doit être différente de celles des autres. Nous lui donnerons la valeur -1 et le représenterons par la chaîne « `\$` ».

Ce symbole sera considéré avec une fréquence d'apparition nulle (il n'apparaît pas dans le texte de départ). L'algorithme d'Huffman produit alors l'arbre de la figure 17. Il est bien sûr différent de celui présenté à la figure 4 puisque nous avons un symbole supplémentaire. Nous en déduisons la table de Huffman (figure 18) et le codage du contenu du fichier (figure 19).

2.3 Représentation de l'arbre

Pour pouvoir analyser le fichier compressé, il faut connaître le codage de Huffman utilisé lors de la compression. Nous allons donc le stoker dans le fichier compressé. Nous commencerons par stocker les symboles qui sont utilisés dans le texte dans leur ordre d'apparition quand on réalise un parcours infixe [1] de l'arbre avec une petite adaptation. En effet, si tous les symboles du texte d'origine pourront être écrits sur un octet, ce n'est pas le cas du symbole de fin de fichier. Aussi, nous écrirons tous les symboles suivant le parcours infixe sauf le symbole de fin de fichier et, avant d'écrire ces symboles, nous écrirons en début de fichier la position (i.e. l'indice) du symbole de fin de fichier dans cette liste (0 s'il est en première position).

Par exemple, pour l'arbre de la figure 17, on obtient les symboles suivants (on utilise la chaîne de caractères qui les représente pour que la correspondance avec l'arbre soit plus facile) :

```
['m', 'x', '\n', 'l', ' ', 't', 'd', '\$', ':', 'p', 'e']
```

Si on considère leur valeur en tant qu'octet (et donc entier), la même liste s'affiche ainsi :

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

1. Un parcours infixe consiste à parcourir le sous-arbre gauche, traiter le nœud, puis parcourir le sous-arbre droit.

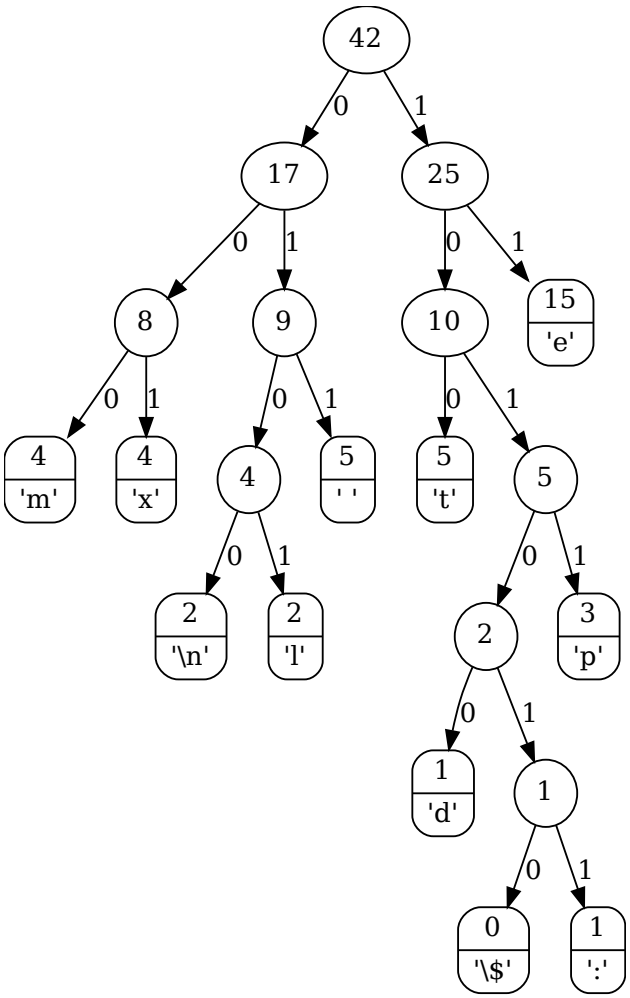

FIGURE 17 – Un arbre de Huffman avec le symbole de fin de fichier

En supprimant le symbole fin de fichier (-1) et en mettant au début sa position, on obtient les octets suivants :

```
[7, 109, 120, 10, 108, 32, 116, 100, 58, 112, 101]
```

Il faut aussi pouvoir détecter la fin des symboles. Dans ce but, on double le dernier symbole. Ainsi, dès que deux symboles consécutifs sont égaux, on sait qu'on a terminé de lire les symboles. Au total, on écrira donc au début du fichier les octets suivants :

```
[7, 109, 120, 10, 108, 32, 116, 100, 58, 112, 101, 101]
```

À la suite des symboles, on code l'arbre sous la forme d'une suite de bits 0 ou 1 correspondant au parcours infixe de l'arbre. On écrit un 0 à chaque fois que l'on descend à gauche et un 1 pour chaque feuille rencontrée. Cet encodage fonctionne ici car nous avons un arbre binaire parfait : chaque nœud a deux sous-arbres (ou aucun pour les feuilles). Pour l'arbre de la figure 17, on aura donc la suite de bits suivante :

```
000110011100100101111
```

Caractère	\$	\n	␣	:	d	e	l	m	p	t	x
Code	101010	0100	011	101011	10100	11	0101	000	1011	100	001

FIGURE 18 – Codage de Huffman avec le symbole de fin de fichier

```
11.001.11.000.1011.0101.11.011.10100.11.011.100.11.001.100.11.011.101011.
↪  0100.11.001.11.000.1011.100.11.011.100.11.000.1011.11.100.11.011.0101.
↪  11.001.11.000.11.0100.101010
```

FIGURE 19 – Codage du texte d'origine (figure 1) en utilisant le code de la figure 18

2.4 Octets et bits

Un octet est un entier compris entre 0 et 255. Un bit est un entier qui vaut soit 0 soit 1.

Pour afficher un entier en base 2, on peut utiliser le paramètre `Base` de la méthode `Put` de `Ada.Integer_Text_IO`.

En Ada, on peut définir un type Octet de la manière suivante.

```
type T_Octet is mod 256;    -- On peut remplacer 256 par 2 ** 8.
```

Ce sont des entiers naturels compris entre 0 et 255. Contrairement aux entiers classiques d'Ada, les entiers modulo ne provoquent pas d'exception en cas de débordement lors des calculs : les calculs se font modulo l'entier naturel précisé. Ainsi, 255 + 1 donne 0.

Les opérateurs `and`, `or` et `xor` sont définis sur ces entiers et correspondent à des opérations réalisées bit à bit.

```
146 and 156 == 144       #     10010010     10010010       10010010   146
146 or  156 == 158       # and 10011100  or 10011100   xor 10011100   156
146 xor 156 ==  14       #   = 10010000   = 10011110     = 00001110
```

Enfin, on peut décaler les bits à droite ou à gauche en faisant des multiplication ou des division par 2.

Pour récupérer le bit de poids fort (le plus à gauche), on peut faire : `Octet / 128` (si `Octet` est du type `T_Octet`). On obtient ici 1. Pour « effacer » ce bit, on décale à gauche en multipliant par 2. Ainsi, pour récupérer successivement les bits d'un octets, on pourra faire :

```
Octet := 146;               -- 10010010
Bit := Octet / 128;
pragma Assert (1 = Bit);
Octet := (Octet * 2);       -- 00100100
Bit := Octet / 128;
pragma assert (0 = Bit);
...
```

Pour reconstruire un octet à partir de 8 bits, on peut appliquer le schéma de Horner. On commence par le bit de poids fort. Avant d'ajouter un nouveau bit, on décale à gauche l'octet puis on fait un ou bit à bit pour ajouter le bit.

```
Octet := 0;                    -- 00000000
Octet := (Octet * 2) or 1;     -- 00000001
Octet := (Octet * 2) or 0;     -- 00000010
Octet := (Octet * 2) or 0;     -- 00000100
Octet := (Octet * 2) or 1;     -- 00001001
Octet := (Octet * 2) or 0;     -- 00010010
Octet := (Octet * 2) or 0;     -- 00100100
Octet := (Octet * 2) or 1;     -- 01001001
Octet := (Octet * 2) or 0;     -- 10010010
pragma Assert (Octet = 146);
```

Dans un fichier, on ne peut pas écrire des bits. On ne peut écrire que des octets. Il faut donc regrouper 8 bits dans un octet puis écrire cet octet dans le fichier.

3 Interface utilisateur.

L'interface avec l'utilisateur se fera au moyen de la ligne de commande.

On écrira deux programmes. Le premier permettra de compresser tous les fichiers fournis sur la ligne de commande. Le fichier compressé aura l'extension .hff. Ainsi, si on demande à compresser le fichier `exemple-sujet.txt`, on obtiendra le fichier `exemple-sujet.txt.hff`. On le fera en tapant la commande suivante :

```
./compresser exemple-sujet.txt
```

Le deuxième programme permettra de décompresser les fichiers fournis sur la ligne de commande s'ils ont l'extension .hff. Les fichiers qui n'ont pas l'extension .hff seront ignorés et un message le signalera. Le fichier décompressé sera engendré dans un fichier de même nom sans l'extension .hff. Ainsi, si on demande à décompresser le fichier `exemple-sujet.txt.hff`, le résultat sera dans le fichier `exemple-sujet.txt`. Voici la commande qui sera utilisée :

```
./decompresser exemple-sujet.txt.hff
```

Si on utilise l'option `-b` (ou `--bavard`) :

```
./compresser -b exemple-sujet.txt
```

le programme devra afficher une trace des principales opérations réalisées de manière à suivre son exécution. Par exemple, pour la compression, on affichera au moins l'arbre de Huffman construit comme présenté à la figure 20 et la table de Huffman (figure 21).

```
(42)
  \--0--(17)
  |      \--0--(8)
  |      |      \--0--(4) 'm'
  |      |      \--1--(4) 'x'
  |      \--1--(9)
  |            \--0--(4)
  |            |      \--0--(2) '\n'
  |            |      \--1--(2) 'l'
  |            \--1--(5) ' '
  \--1--(25)
        \--0--(10)
        |      \--0--(5) 't'
        |      \--1--(5)
        |            \--0--(2)
        |            |      \--0--(1) 'd'
        |            |      \--1--(1)
        |            |            \--0--(0) '\$'
        |            |            \--1--(1) ':'
        |            \--1--(3) 'p'
        \--1--(15) 'e'
```

FIGURE 20 – Version textuelle de l'arbre de Huffman de la figure 17

```
'\$' --> 101010
'\n' --> 0100
' '  --> 011
':'  --> 101011
'd'  --> 10100
'e'  --> 11
'l'  --> 0101
'm'  --> 000
'p'  --> 1011
't'  --> 100
'x'  --> 001
```

FIGURE 21 – Version textuelle de la table de Huffman de la figure 18

4 Exigences pour la réalisation du projet

1. Le projet se fera en équipes de 2 du même groupe de TD. Les deux membres de l'équipe ne doivent avoir été ensemble pour le mini-projet 1.
   Si un étudiant n'a pas d'équipe, il doit en informer rapidement son enseignant de TD. Il pourra alors faire équipe avec un étudiant seul d'un autre TD du même CM.
2. Le langage utilisé est le langage Ada limité aux concepts vus en cours, TD ou TP et aux éléments présentés dans ce sujet et les exemples fournis.
3. Le programme devra compiler et fonctionner sur les machines Linux de l'N7.
4. L'ensemble des concepts du cours devront être mis en œuvre, en particulier :
   — Écriture des spécifications pour tous les programmes et sous-programmes
   — Conception en utilisant la méthode des raffinages
   — Justification des choix des types de données manipulés
   — Conception de modules, de préférence réutilisables : encapsulation, généricité, TAD...
   — Définition des tests et le processus de test mis en place
5. SVN sera utilisé pour rendre le travail fourni. Vos modifications seront poussées régulièrement pour montrer que vous avez bien suivi toutes les étapes recommandées pour la réalisation de votre projet. Les versions intermédiaires ne seront pas prises en compte dans la notation. Elles pourront toutefois démontrer votre implication dans le projet.
   Votre dépôt SVN sera disponible à l'URL suivante :

   ```
   http://cregut.svn.enseeiht.fr/2021/1sn/pim/projet/XXX
   ```

   où XXX est à remplacer par le numéro de votre équipe de projet (voir l'activité constitution des équipes du projet sur Moodle).
   Les dépôts ne pourront être créés que quand les équipes seront constituées.
6. Les développements associés au projet comprendront les dossiers suivants.
   — Le dossier « livrables » ne devra contenir que les livrables explicitement demandés.
   — Le dossier « src » contiendra les sources de votre application.
   — Le dossier « doc » pourra être utilisé pour la rédaction du rapport.
   — Vous pouvez bien sûr créer d'autres dossiers si vous en avez besoin.

5 Livrables et échéances

— jeudi 18 novembre : **constitution des équipes**.
— samedi 4 décembre : **raffinages des deux programmes demandés, principaux types de données et structuration en modules**.

Ce livrable prendra la forme d'un document partagé (Google Doc) appelé PIM-Huffman-raffinages-XXX (XXX est le nom de votre équipe) dont la version PDF sera déposée dans le dossier `livrables`.

La grille d'évaluation des raffinages qui avait été utilisée dans le mini-projet 1 devra être complétée pour les deux raffinages proposés (compresser et décompresser).

— samedi 11 décembre : **interfaces des modules et programmes de tests associés**.

Les interfaces des modules (`*.ads`) et les programmes de test (`test_*.adb`) seront écrits dans le dossier src/ (votre répertoire de travail pour le projet). Une copie de ces fichiers sera déposée dans le dossier `livrables/interfaces`.

— samedi 15 janvier : rendu des **sources** (archive [2] nommée `sources.tgz`), du **manuel utilisateur** (`manuel.pdf`), du **rapport** du projet (`rapport.pdf`), du **script de la démonstration** (`demo.txt` ou `demo.pdf`) et de la **présentation orale** (`presentation.pdf`) dans le dossier `livrables`.

Le **manuel utilisateur** décrit comment utiliser les programmes développés. Il est illustré avec des copies d'écran.

Le **rapport** doit au moins contenir les informations suivantes :

- un résumé qui décrit l'objectif et le contenu du rapport (10 lignes maxi),
- une introduction qui présente le problème traité [3] et le plan du document,
- l'architecture de l'application en modules,
- la présentation des principaux choix réalisés,
- la présentation des principaux algorithmes et types de données,
- la démarche adoptée pour tester le programme,
- les difficultés rencontrées et les solutions adoptées en justifiant vos choix (en particulier quand vous avez envisagé plusieurs solutions),
- l'organisation de l'équipe (qui a fait quoi, etc.),
- un bilan technique donnant un état d'avancement du projet et les perspectives d'amélioration / évolution,
- une bilan *personnel* et *individuel* : intérêt, temps passé, temps passé à la conception, temps passé à l'implantation, temps passé à la mise au point, temps passé sur le rapport, enseignements tirés de ce projet, etc.

Remarque : Bien sûr, les modifications faites par rapport aux premiers livrables (raffinages, structures de données, interfaces des modules, programmes de tests) devront être indiquées et argumentées dans le rapport.

Le script de la démonstration et la présentation sont décrits dans l'item suivant.

2. On utilisera la commande `tar` pour produire cette archive. Par exemple, on pourra faire : `tar czvf sources.tgz *.ad[sb] *.txt`.

3. La présentation du problème doit être concise car les enseignants connaissent le sujet !

— Semaine du 17 janvier : **Recette du projet**

La recette du projet durera environ 15 minutes par équipe. Elle sera organisée en 3 phases : une démonstration, une présentation orale et un échange avec l'enseignant.

La démonstration durera 4 minutes. Vous avez la main. Il s'agit de montrer que vos programmes répondent au cahier des charges. Cette démonstration doit être préparée. Le **script de la démonstration** décrit le déroulé de la démonstration et explique ce que vous allez montrer. Il doit être rendu sur SVN sous la forme d'un fichier `demo.txt` ou `demo.pdf`.

La présentation dure 4 minutes également. Il s'agit de nous expliquer les principaux choix fait dans votre projet : architecture en modules, structures de données, principaux algorithmes, avancement du projet... Un support de **présentation** doit être préparé et rendu (`presentation.pdf`).

L'échange dure environ 7 minutes. Il s'agit de questions/réponses, discussions, tests supplémentaires...

L'ordre de passage sera communiqué via Moodle.

6 Principaux critères de notation

Voici quelques uns des critères qui seront pris en compte lors de la notation du projet :

— le respect du cahier des charges,

— la qualité des raffinages,

— la facilité à comprendre la solution proposée,

— la pertinence des modules définis,

— la pertinence des sous-programmes définis,

— la qualité des sous-programmes : ils ne doivent pas être trop longs, ne pas faire trop de choses, ne pas avoir trop de structures de contrôle imbriquées. Dans ces cas, il faut envisager de découper le sous-programme !

— la bonne utilisation de la programmation par contrat (en particulier les préconditions et les invariants de type) et de la programmation défensive,

— la pertinence des tests réalisés sur les sous-programmes

— la pertinence des types utilisateurs définis,

— la pertinence de l'architecture logicielle adoptée,

— l'absence de code redondant,

— le choix des identifiants,

— les commentaires issus des raffinages,

— la bonne utilisation des commentaires,

— le respect de la solution algorithmique (les raffinages) dans le programme,

— la présentation du code (le programme doit être facile à lire et à comprendre),

— l'utilisation des structures de contrôle adéquates,
— la validité du programme,
— la robustesse du programme,
— la bonne utilisation de la mémoire dynamique (valgrind).

Attention : Cette liste n'est pas limitative !

7 Barème indicatif pour l'évaluation

1. Raffinages (4 points)
   — respect du formalisme présenté
   — qualité du raffinage
   — pertinence
   — respectés dans le code
2. Réalisation (6 points)
   — organisation en modules
   — caractère réutilisable des modules (généricité, complétude des opérations)
   — modélisation des données (types utilisés)
   — algorithmes
   — qualité du code
3. Fonctionnement du programme (4 points)
   — couverture du sujet
   — tests réalisés (y compris unitaires)
4. Démonstration (1.5 points)
   — préparation de la démonstration
   — couverture du sujet
5. Présentation orale (1.5 points)
   — qualité de la présentation
   — couverture du sujet
6. Rapport (3 points)
   — structure générale (introduction, conclusion, plan)
   — qualité informative du rapport (raffinages, types, état d'avancement, difficultés)
   — qualité de la présentation (forme, grammaire, orthographe...)

8 Indications

Voici quelques questions auxquelles vous devrez apporter des réponses dans le rapport (et les programmes) que vous rendrez...

1. Comment représenter l'arbre de Huffman ?
2. Est-ce que le codage d'un caractère est toujours composé d'au plus 8 bits ?

3. Comment représenter le code d'un caractère ?
4. Comment représenter la table de Huffman ?
5. La position du symbole de fin de fichier peut-elle toujours être représentée sur un octet ?
6. Lors de la décompression, comment reconstruire l'arbre de Huffman ?
7. Lors de la décompression, comment faire pour décoder les symboles du texte d'origine ?