Codage de Huffman

Table des matières

1. Le principe..2
2. L'Algorithme de compression de Huffman..2
    2.1. Convention de codage binaire..2
    2.2. Codage à taille fixe, à taille variable..3
    2.3. Le codage à taille variable..4
3. L'algorithme de Huffman..4
    3.1. Les arbres binaires..5
    3.2. Implémentation de l'algorithme de Huffman..5
    3.3. Exemple..6
    3.4. Code source..9

Le codage de Huffman est une méthode de compression statistique de données qui permet de réduire la longueur du codage d'un alphabet. Le code de Huffman (1952) est un code de longueur variable optimal, c'est-à-dire tel que la longueur moyenne d'un texte codé soit minimale. On observe ainsi des réductions de taille de l'ordre de 20 à 90%.

1. Le principe

Le principe de l'algorithme de Huffman consiste à recoder les octets rencontrés dans un ensemble de données source avec des valeurs de longueur binaire variable.

L'unité de traitement est ramenée au bit. Huffman propose de recoder les données qui ont une occurrence très faible sur une longueur binaire supérieure à la moyenne, et recoder les données très fréquentes sur une longueur binaire très courte.

Ainsi, pour les données rares, nous perdons quelques bits regagnés pour les données répétitives. Par exemple, dans un fichier ASCII le "w" apparaissant 10 fois aura un code très long: 0101000001000. Ici la perte est de 40 bits (10 x 4 bits), car sans compression, il serait codé sur 8 bits au lieu de 12. Par contre, le caractère le plus fréquent comme le "e" avec 200 apparitions sera codé par 1. Le gain sera de 1400 bits (7 x 200 bits). On comprend l'intérêt d'une telle méthode.

De plus, le codage de Huffman a une propriété de préfixe : une séquence binaire ne peut jamais être à la fois représentative d'un élément codé et constituer le début du code d'un autre élément.

Si un caractère est représenté par la combinaison binaire 100 alors la combinaison 10001 ne peut être le code d'aucune autre information. Dans ce cas, l'algorithme de décodage interpréterait les 5 bits comme une succession du caractère codé 100 puis du caractère codé 01. Cette caractéristique du codage de Huffman permet une codification à l'aide d'une structure d'arbre binaire.

2. L'Algorithme de compression de Huffman

Cet algorithme est "non-destructif", c'est à dire qu'il compresse les données sans introduire de perte d'information, de sorte que le fichier décompressé est une copie conforme de l'original.

Le codage employé est dit "entropique", c'est à dire qu'il se base sur les statistiques d'apparition des différents octets du fichier, et "à taille variable", c'est à dire que chaque octet est codé selon une suite de bits, dont la taille diffère selon l'octet.

Pour comprendre en quoi un codage "entropique" peut nous permettre de compresser un fichier, nous commençons cette présentation par un rappel sur le principe du stockage d'informations en binaire.

2.1. Convention de codage binaire

On code un nombre entier compris entre 0 et 255 par une séquence de 8 chiffres binaires (binary digits, ou "bits" en anglais, valant 0 ou 1), encore appelée octet.

A priori, il suffit pour cela de se donner une table de correspondance du genre:

...	...
138	10001010
139	10001011
...	...

La correspondance peut être quelconque, du moment qu'à chaque entier entre 0 et 255 est affecté un code binaire et un seul. Une fois une correspondance fixée, il suffit de la prendre comme convention.

En pratique, la convention qui a été choisie est celle de la représentation en base 2 qui a le mérite d'être naturelle et logique. Dans la table ci dessus, 10001010 est la représentation en base 2 de 138. C'est la manière de noter les chiffres que nous aurions adoptée si nous n'avions que deux doigts au lieu de dix.

L'octet défini selon cette convention est l'unité de base de stockage des données. Tout fichier informatique est une suite d'octets rangés selon un ordre bien défini. La taille du fichier est simplement le nombre d'octets qui le constituent. Le kilo-octet (Ko) correspond à 1024 (et non pas 1000) octets, le méga-octet (Mo) à 1024*1024 octets.

2.2. Codage à taille fixe, à taille variable

La problématique de la compression "entropique" de données repose sur une AUTRE manière de coder les chiffres, peut-être moins logique mais plus judicieuse, de telle manière que la taille du même fichier recidé selon la nouvelle convention, serait MOINS GRANDE?

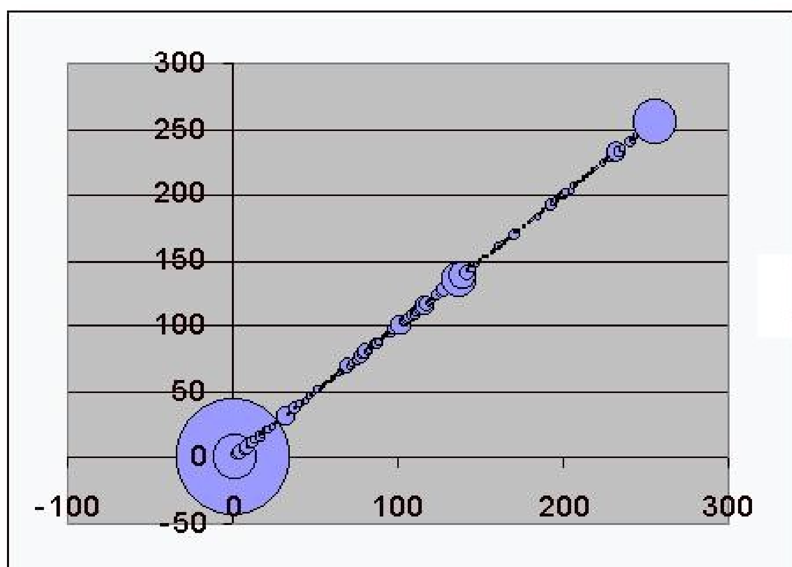
Dans les textes longs, les lettres n'apparaissent pas avec la même fréquence. Ces fréquences varient suivant la langue utilisée. En français, les lettres les plus rencontrées sont dans l'ordre :

E S A I N T R U L O D C P M V Q G F H B X J Y Z K W

avec les fréquences (souvent proches et dépendant de l'échantillon utilisé) :

E	S	A	I	N	T	R	U	L	O	D
14.69%	8.01%	7.54%	7.18%	6.89%	6.88%	6.49%	6.12%	5.63%	5.29%	3.66%

A titre d'exemple, voici l'analyse d'un contenu d'octets du fichier Wordpad.exe. Dans ce tableau, en abscisse comme en ordonnée sont portées les valeurs possibles d'un octet donné (donc 0 ... 255). Sur la diagonale, à l'abscisse et ordonnée correspondante la taille du cercle est proportionnelle au nombre d'octets ayant cette valeur dans le fichier.



Valeur	Nombre	Fréquence
0	71891	34.4%
2	1119	0.53%
128	1968	0.94%
130	79	0.038%
255	10422	4.99%

On voit clairement que les valeurs 0, 128 et 255 sont nettement plus fréquentes que les autres!

<u>L'idée générale du codage entropique est la suivante:</u>

On va décider d'une convention de codage "à taille variable", qui représente une valeur fréquente PAR UN PETIT NOMBRE DE BITS, et une valeur peu fréquente par un grand nombre de bits.

Par exemple, 0 sera maintenant représenté par la séquence "11" (anciennement "00000000"), 128 par "1011010" (anciennement "10000000"), 255 par "0100" (anciennement "11111111"), etc...

Étant donné que 0 représente à lui seul un tiers du fichier, on a gagné une place considérable en le codant sur deux bits au lieu de huit ! Et idem pour les autres valeurs fréquentes...

2.3. Le codage à taille variable

Comment décoder le fichier compressé ?

Supposons que nous décidons d'une convention de code à taille variable, qui fait correspondre, entre autres, les valeurs suivantes :

0	"11"
2	"11010"
12	"00"
127	"0111100"
255	"0100"

Supposons alors que nous ayons à décoder la séquence : 1101000111100

Plusieurs interprétations sont possibles :

1101000111100 = 11 0100 0111100 = 0 255 127

1101000111100 = 11010 00 11 11 00 = 2 12 0 0 12

Avec plusieurs possibilités équivalentes, on est incapable de retranscrire le code initial.

Le problème qui s'est posé ici est que CERTAINS CODES sont le DEBUT D'AUTRES CODES. Ici, "11" est le code du nombre 0, mais c'est aussi le début de "11010", code du nombre 2. D'où l'ambiguïté.

On appelle "code-préfixe" un codage dans lequel aucun code n'est le début d'un autre.

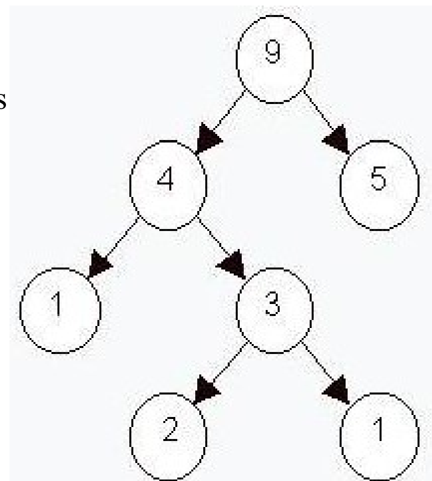
Pour qu'il n'y ait aucune ambiguïté au moment du décodage, il nous faut absolument un code-préfixe.

3. L'algorithme de Huffman

L'algorithme de Huffman génére un code-préfixe à taille variable, à partir de la table des fréquences d'une suite de valeurs.

3.1. Les arbres binaires

On appelle arbre binaire une collection d'éléments (les "nœuds") dans laquelle chaque nœud est connecté à 0 ou 2 autres nœuds. A chaque nœud on peut associer une ou plusieurs valeurs. Voici un exemple d'arbre binaire:



L'élément supérieur de l'arbre (le "9") est la racine; tandis que les nœuds terminaux sont les feuilles.

3.2. Implémentation de l'algorithme de Huffman

Soit un fichier à compresser.

Au préalable, dresser la table des fréquences (Clé / Valeur) ci-contre.

Dans ce tableau, Fréquence(i) désigne le nombre d'octets du fichier ayant la valeur i.

Clé	Valeur
0	Fréquence(0)
1	Fréquence(1)
...	
255	Fréquence(255)

Pour chaque Clé associée à une fréquence non-nulle, créer un nœud terminal, et affectez lui le couple (Clé, Valeur). Chaque nœud terminal créé représente autant d'arbre binaire à élément unique.

Itérer :

1. Classer chaque arbre binaire par Valeur croissante;
2. Retirer de la liste les deux arbres binaires ayant les Valeurs les plus faibles. Insérer, à la place, un nouvel arbre binaire dont la racine pointe sur les racines des deux arbres binaires supprimés. A ce nouvel arbre binaire, affectez lui pour Valeur, la somme des Valeurs des deux arbres précédents. Inutile de lui affecter une Clé.
3. Recommencer en 1 jusqu'à ce qu'il ne reste plus qu'un arbre binaire unique.

On construit ainsi un arbre binaire dans lequel les nœuds terminaux sont les éléments de notre table des fréquences. Pour connaître le Code de Huffman associé à chaque Clé, il suffit de descendre l'arbre, en partant de la racine, pour rejoindre la Clé voulue. A chaque embranchement, on compte "0" si l'on est passé à gauche, et "1" si l'on est passé à droite.

3.3. Exemple

Supposons que notre fichier soit extrêmement simple, et constitué d'un mot unique :

anticonstitutionnellement

Il y a 25 caractères dans ce fichier; chaque caractère étant codé par un octet de 8 bits (codage ASCII) cela signifie donc 25 octets, ou encore 200 bits.

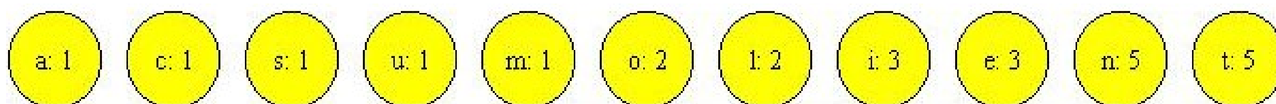
Table des fréquences:

Clé	a	c	s	u	m	o	l	i	e	n	t
Fréquence	1	1	1	1	1	2	2	3	3	5	5

Tous les autres octets dont la fréquence est nulle ne sont pas représentés.

Créer un nœud terminal pour chaque entrée du tableau:

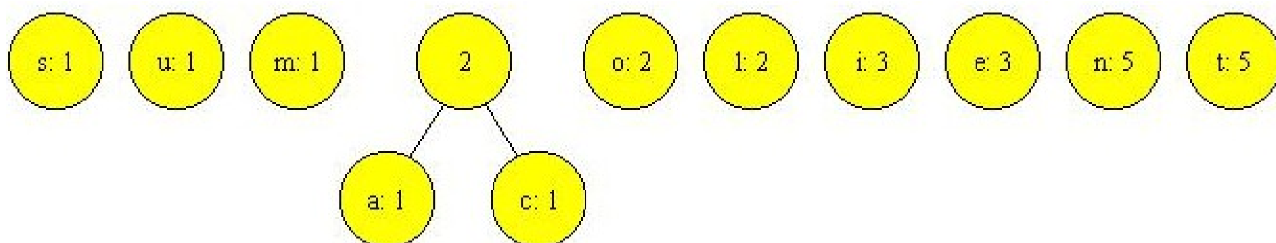
Étape 0:



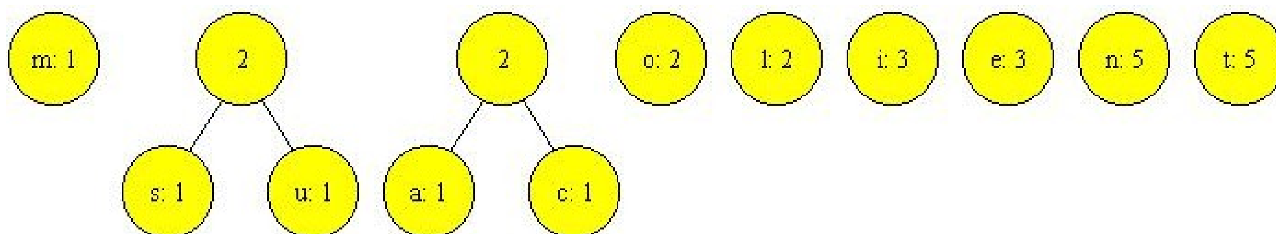
Ce qui nous fait pour l'instant 11 arbres contenant un seul nœud chacun.

On commence l'itération: à chaque fois on supprime les deux arbres de gauche et on les remplace par un "arbre somme". Le nouvel arbre est inséré dans la liste en respectant l'ordre croissant, et on recommence jusqu'à ce qu'il ne reste plus qu'un seul arbre :

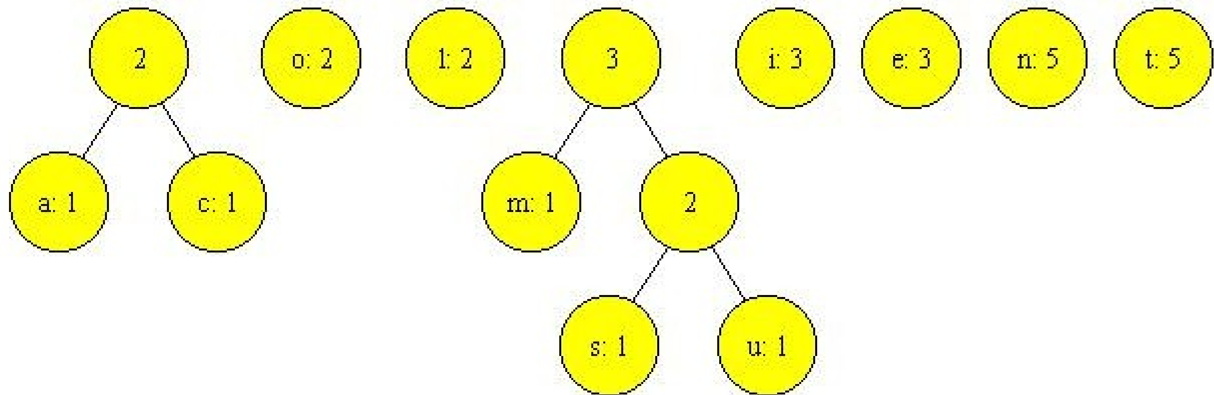
Étape 1:



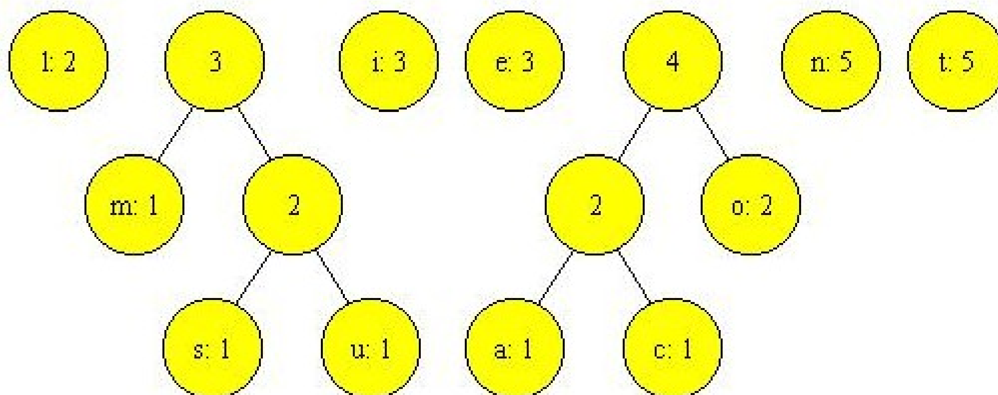
Étape 2:



Étape 3:

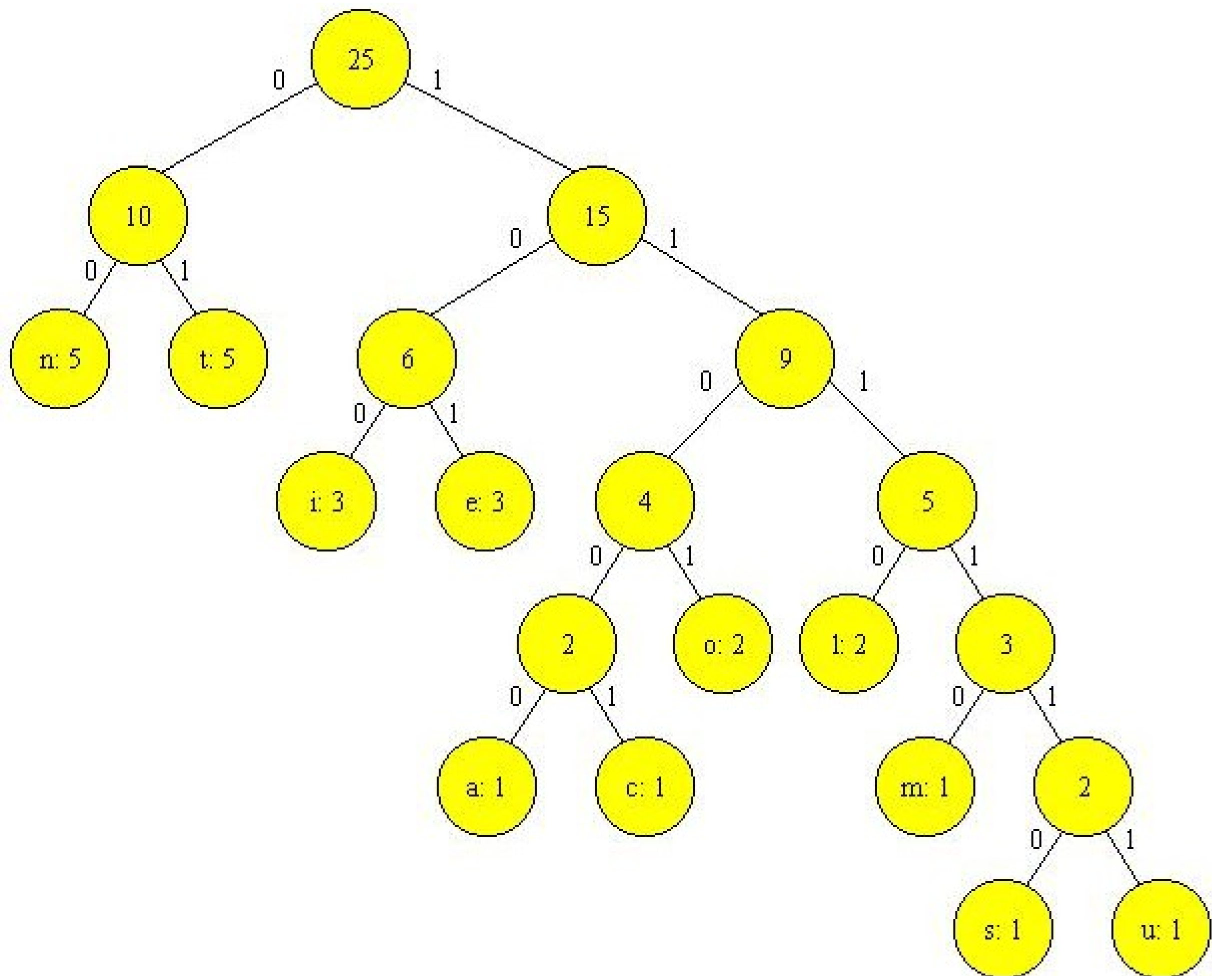


Étape 4:



etc ...

L'arbre final:



Maintenant, le code associé à chaque Clé n'est autre que le chemin d'accès au nœud terminal correspondant, depuis la racine, en notant 0 la branche de gauche et 1 la branche de droite.

Finalement :

Clé	Code binaire
n	00
t	01
i	100
e	101
a	11000
c	11001
o	1101
l	1110
m	11110
s	111110
u	111111

Et voici maintenant, transcrit avec notre nouveau code, le mot de départ:

110000001100110011101001111100110001111111011001101000010111101110101111101010001

ce qui nous fait 81 bits, au lieu de 200 au départ ! Cela correspond à un taux de compression de 60 %.

Le fait d'avoir généré un code en se servant d'un arbre binaire assure qu'aucun code ne peut être le préfixe d'un autre. Vous pouvez vérifier qu'à l'aide de la table de codage, il n'y a aucune ambiguïté possible pour décoder le mot compressé.

En pratique, la table de codage étant spécifique à chaque fichier, il est indispensable de l'incorporer au fichier compressé, de manière à ce que le décryptage soit possible. Ce qui signifie que la taille du fichier compressé doit être augmentée d'autant. Dans le cas de notre fichier exemple, il faudrait incorporer AU MINIMUM 22 octets de plus pour insérer la table de codage, et le taux de compression n'est plus aussi bon.

Toutefois, pour des fichiers suffisamment larges (à partir de quelques kilo-octets) le surplus de taille généré par la table de codage devient négligeable par rapport à l'ensemble du fichier. Concrètement, l'algorithme de Huffman permet d'obtenir des taux de compression typiques compris entre 30% et 60% (sans être aussi bien que ce que réalise Winzip, en moyenne 20 % de mieux).

3.4. Code source

Ce programme permet de compresser et décompresser un fichier en utilisant l'algorithme de Huffman. Les taux de compression obtenus sont bien évidemment minables par rapport à zip ou rar, mais bon, comme on dit, l'important c'est de participer !

Syntaxe :

huffman c/d/f source destination [frequence]
- c pour compresser le fichier source dans destination.
- d pour le décompresser
- f pour créer un fichier de fréquences

L'option frequence permet de spécifier un fichier de fréquences qui sera utilisé pour compresser le fichier source.

```
/*
Outil de compactage/décompactage de fichier utilisant le codage de Huffman
Compilation sous Linux : gcc huffman.c -o huffman
*/

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

#define CODE_MAX_LEN 32
#define BUF_LEN 512

/* Definition des structures */
struct arbre
{
   short branche0;
   short branche1;
};
```

```
struct arbre_data
{
   unsigned long frequence;
   short index_suivant;
};

struct dictionnaire
{
   unsigned char taille;
   char code[CODE_MAX_LEN];
};

/* Prototypes des fonctions */
short huffman_calculer_frequences(FILE *, unsigned long *, unsigned short *);
short huffman_lire_frequences(FILE *);
short huffman_creer_arbre(short);
void huffman_creer_dictionnaire(unsigned char *, short, short);
char huffman_creer_fichier_frequences(FILE *, FILE *);
char huffman_compacter(FILE *, FILE *, FILE *);
char huffman_decompacter(FILE *, FILE *, FILE *);

/* Variables globales */
struct arbre_data *arbre_d;
struct arbre *arbre;
struct dictionnaire *dico;

/*
Calcule la fréquence de chaque caractère dans un fichier et trie la liste de structures
par fréquences croissantes
*src : pointeur sur le fichier
*nbre_octets : pointeur sur une variable recevant la taille du fichier
*nbre_ascii : pointeur sur une variable recevant le nombre de caractères
différents présents dans le fichier source
Retourne l'index de la première structure dans la liste triée
*/
short huffman_calculer_frequences(FILE *src, unsigned long *nbre_octets, unsigned short
*nbre_ascii)
{
   int i, r;
   unsigned char buffer[BUF_LEN];
   short index_frequence=0, index_precedent=-1;

   int continuer=1;
   short c1, c2;

   *nbre_octets=0;
   *nbre_ascii=0;

   memset(arbre_d, 0, 512*sizeof(struct arbre_data));

   /*Lecture dans le fichier */
   while ((r=fread(buffer, 1, BUF_LEN, src))>0)
   {
      *nbre_octets+=r;
      for(i=0; i<r; i++)
         /* incrémentation du compteur du caractère correspondant */
```

```
      arbre_d[buffer[i]].frequence++;
   }

   /* Chainage des structures avec une fréquence supérieure à 0 */
   for(i=0; i<256; i++)
      if (arbre_d[i].frequence>0)
      {
         (*nbre_ascii)++;
         if (index_precedent==-1)
            index_frequence=i;
         else
            arbre_d[index_precedent].index_suivant=i;
         index_precedent=i;
      }
   if (index_precedent==-1)
      index_frequence=-1;
     else
      arbre_d[index_precedent].index_suivant=-1;

   /* Tri des structures (bubble sort) */
   while (continuer)
   {
      c1=index_frequence;
      continuer=0;
      index_precedent=-1;
      while(c1!=-1)
      {
         if ((c2=arbre_d[c1].index_suivant)!=-1)
         {
            if (arbre_d[c1].frequence>arbre_d[c2].frequence)
            {
               continuer=1;
               if (index_precedent==-1)
                  index_frequence=c2;
               else
                  arbre_d[index_precedent].index_suivant=c2;
               arbre_d[c1].index_suivant=arbre_d[c2].index_suivant;
               arbre_d[c2].index_suivant=c1;
            }
            index_precedent=c1;
            c1=c2;
         }
         else
            c1=c2;
      }
   }

   /* On retourne l'index de la première structure */
   return index_frequence;
}

/*
Lit les fréquences de chaque caractère, inscrites soit dans le fichier
compressé, soit dans un fichier de fréquences spécial
*/
short huffman_lire_frequences(FILE *frq)
{
```

```
   unsigned short nbre_ascii;
   unsigned char i;
   short index_frequence=-1, index_precedent=-1;

   /* Lecture de la taille de la table des fréquences */
   fread(&nbre_ascii, 2, 1, frq);

   /* Lecture de la table des fréquences */
   while (nbre_ascii>0)
   {
      /* Lecture du caractère en cours */
      fread(&i, 1, 1, frq);
      /* Lecture de la fréquence du caractère en cours */
      fread((char *)&arbre_d[i].frequence, 4, 1, frq);
      /* Chainage de la structure */
      if (index_frequence==-1)
         index_frequence=i;
      else
         arbre_d[index_precedent].index_suivant=i;
      index_precedent=i;
      nbre_ascii--;
   }

   if (index_precedent==-1)
      return -1;

   arbre_d[index_precedent].index_suivant=-1;
   return index_frequence;
}

/*
Crée un arbre à partir d'une liste de fréquences
index_fréquence : idnex de la première structure dans la liste *arbre_d
Retourne l'index de la racine de l'arbre dans la liste *arbre
*/
short huffman_creer_arbre(short index_frequence)
{
   short i, j, j_save;
   unsigned long somme_frequence;
   short nbre_noeuds=256;
   char struct_inseree=0;

   /* Les structures 0 à 255 correspondent aux caractères, ce sont des terminaisons =>
-1 */
   for(j=0; j<256; j++)
      arbre[j].branche0=arbre[j].branche1=-1;

   /* Création de l'arbre :
   La mise en commun les deux fréquences les plus faibles crée un nouveau noeud avec
une frequence
   égale a la somme des deux fréquences.
   Il s'agit ensuite d'insérer cette nouvelle structure dans la liste triée */
   i=index_frequence;
   while(i!=-1)
   {
      if (arbre_d[i].index_suivant==-1)
      {
```

```
           /*printf("Arbre cree\n");*/
           break;
        }

          #ifdef DEBUG
          printf("%d\n", arbre_d[i].frequence);
          printf("%d\n", arbre_d[arbre_d[i].index_suivant].frequence);
          #endif // DEBUG

        somme_frequence=arbre_d[i].frequence +
arbre_d[arbre_d[i].index_suivant].frequence;

          #ifdef DEBUG
          printf("Nouveau noeud : %d (%d) et %d (%d) => %d\n",
                 i,
                 arbre_d[i].frequence, arbre_d[i].index_suivant,
arbre_d[arbre_d[i].index_suivant].frequence,
                  somme_frequence);
          #endif // DEBUG

        arbre_d[nbre_noeuds].frequence=somme_frequence;
        arbre[nbre_noeuds].branche0=arbre_d[i].index_suivant;
        arbre[nbre_noeuds].branche1=i;

        /* Insertion du nouveau noeud dans la liste triée */
        j_save=-1;
        struct_inseree=0;
        j=i;
        while (j!=-1 && struct_inseree==0)
        {
           if (arbre_d[j].frequence>=somme_frequence)
           {
              if (j_save!=-1)
                 arbre_d[j_save].index_suivant=nbre_noeuds;
              arbre_d[nbre_noeuds].index_suivant=j;

              #ifdef DEBUG
              printf("Insertion du nouveau noeud : entre %d et %d\n",
                  j_save==-1?-1:arbre_d[j_save].frequence, arbre_d[j].frequence);
              #endif // DEBUG

              struct_inseree=1;
           }
           j_save=j;
           j=arbre_d[j].index_suivant;
        }
        /* Insertion du nouveau noeud a la fin */
        if (struct_inseree==0)
        {
           arbre_d[j_save].index_suivant=nbre_noeuds;
           arbre_d[nbre_noeuds].index_suivant=-1;

              #ifdef DEBUG
              printf("Insertion du nouveau noeud à la fin : %d\n",
arbre_d[j_save].frequence);
              #endif // DEBUG
        }
```

```
      nbre_noeuds++;
      i=arbre_d[i].index_suivant;
      i=arbre_d[i].index_suivant;
   }
   /* On retourne l'index du noeud racine */
   return nbre_noeuds-1;
}

/*
Procédure récursive qui crée un dictionnaire (correspondance entre la valeur ascii
d'un caractère et son codage obtenu avec la compression huffman) à partir d'un arbre
*code : pointeur sur une zone mémoire de taille CODE_MAX_LEN recevant
temporairement le code, au fur et à mesure de la progression dans l'arbre
index : position dans l'arbre (index de la structure courante)
pos : nombre de bits deja inscrits dans *code
*/
void huffman_creer_dictionnaire(unsigned char *code, short index, short pos)
{
   /* On a atteint une terminaison de l'arbre : c'est un caractère */
   if ((arbre[index].branche0==-1) && (arbre[index].branche1==-1))
   {
      /* Copie du code dans le dictionnaire */
      memcpy(dico[index].code, code, CODE_MAX_LEN);

        #ifdef DEBUG
        printf("%c: %x - %d\n", index, code[0], pos));
        #endif // DEBUG

      /* taille du code en bits */
      dico[index].taille=(unsigned char)pos;
   }
   /* le noeud possède d'autres branches : on continue à les suivre */
   else
   {
      /* On suit la branche ajoutant un bit valant 0 */
      code[pos/8]&=~(0x80>>(pos%8));
      /* Le "(short)" devant "(pos+1)", c'est juste pour empecher VC++
      de chipoter (Warning : integral size mismatch in argument :
      conversion supplied) */
      huffman_creer_dictionnaire(code, arbre[index].branche0, (short)(pos+1));
      /* On suit la branche ajoutant un bit valant 1 */
      code[pos/8]|=0x80>>(pos%8);
      huffman_creer_dictionnaire(code, arbre[index].branche1, (short)(pos+1));
   }
}

/*
Crée une table de fréquences à partir de src, l'inscrit dans dst,
puis affiche un tableau des caractères de la table, avec leur fréquence.
*/
char huffman_creer_fichier_frequences(FILE *src, FILE *dst)
{
   short i, compteur=0;
   unsigned long nbre_octets;
   unsigned short nbre_ascii=0;

   i=huffman_calculer_frequences(src, &nbre_octets, &nbre_ascii);
```

```c
   /*Ecriture de la taille de la table des fréquences */
   fwrite(&nbre_ascii, 1, 1, dst);

   printf("Creation du fichier de frequences...\n");
   printf("%d caracteres representes sur 256\n", nbre_ascii);

   /*Ecriture de la table des fréquences */
    while(i!=-1)
   {
      nbre_ascii=i;
      fwrite(&nbre_ascii, 1, 1, dst);
      fwrite((char *)&arbre_d[i].frequence, 4, 1, dst);
      if ((nbre_ascii>=32 && nbre_ascii<=126) || (nbre_ascii>=161 && nbre_ascii <=255))
         printf("%-4c %-6d", nbre_ascii, arbre_d[i].frequence);
      else
         printf("0x%02x %-6d", nbre_ascii, arbre_d[i].frequence);
      if (((compteur+1)%7)==0)
         printf("\n");
      compteur++;
      i=arbre_d[i].index_suivant;
   }
   printf("\n");
   return 0;
}

/*
Compresse le fichier src.
Le résulat se trouve dans le fichier dst
Si frq est différent de NULL, il est utilsé pour lire la table des fréquences
nécessaire à la construction de l'arbre
*/
char huffman_compacter(FILE *src, FILE *dst, FILE *frq)
{
   int i, r, octet_r, bit_r, bit_count, bit_w;
   unsigned long nbre_octets;
   short index_frequence;
   short racine_arbre;
   unsigned short nbre_ascii=0;
   unsigned char code[CODE_MAX_LEN];
   unsigned char buffer[BUF_LEN];

   printf("Compression en cours...\n");

   /* création ou lecture de la table des fréquences */
   if (frq==NULL)
   {
      index_frequence=huffman_calculer_frequences(src, &nbre_octets, &nbre_ascii);
   }
   else
   {
      printf("Utilisation d'un fichier de frequences pour la creation du dico\n");
      fseek(src, 0, SEEK_END);
      nbre_octets=ftell(src);
      index_frequence=huffman_lire_frequences(frq);
   }
```

```
/*Ecriture de la taille en octets du fichier original */
fwrite((char *)&nbre_octets, 4, 1, dst);
/*Ecriture de la taille de la table des fréquences */
fwrite(&nbre_ascii, 2, 1, dst);

/*Ecriture de la table des fréquences */
if (frq==NULL)
{
   i=index_frequence;
   while(i!=-1)
   {
      nbre_ascii=i;
      fwrite(&nbre_ascii, 1, 1, dst);
      fwrite((char *)&arbre_d[i].frequence, 4, 1, dst);
      i=arbre_d[i].index_suivant;
   }
}

/* Construction de l'arbre à partir de la table des fréquences */
racine_arbre=huffman_creer_arbre(index_frequence);

/* Allocation de mémoire pour le dictionnaire */
if ((dico=(struct dictionnaire *)malloc(256*sizeof(struct dictionnaire)))==NULL)
{
   /*free(arbre_d);
   free(arbre);*/
   perror("malloc");
   return -1;
}

/* RAZ du champs taille du dico. Si on utilise une table de fréquences
prédéfinie pour la compression, et qu'un caractère à compresser n'est pas
présent dans la table, alors il ne sera pas traité */
for(i=0; i<256; i++)
   dico[i].taille=0;

/* Création du dictionnaire à partir de l'arbre */
huffman_creer_dictionnaire(code, racine_arbre, 0);

/* Compression du fichier source et écriture dans le fichier cible */
fseek(src, 0, SEEK_SET);
code[0]=0;
bit_w=0x80;
/* Lecture de BUF_LEN octets dans le fichier source */
while ((r=fread(buffer, 1, BUF_LEN, src))>0)
{
   /* Traitement octet par octet */
   for(i=0; i<r; i++)
   {
      /* Ecriture du code correspondant au caractère dans le dictionnaire */
      octet_r=0;
      bit_r=0x80;
      /* Ecriture bit par bit */
      for(bit_count=0; bit_count<dico[buffer[i]].taille; bit_count++)
      {
         if (dico[buffer[i]].code[octet_r] & bit_r)
            code[0]|=bit_w;
```

```
            /*else
               code[0]&=~(bit_w); */
            bit_r>>=1;
            if (bit_r==0)
            {
               octet_r++;
               bit_r=0x80;
            }
            bit_w>>=1;
            if (bit_w==0)
            {
               /*printf("%3x", code[0]);*/
               fputc(code[0], dst);
               code[0]=0;
               bit_w=0x80;
            }
         }
      }
   }
   if(bit_w!=0x80)
      fputc(code[0], dst);

   free(dico);

   printf("Compactage termine. Taux de compression : %.2f\n",
(float)nbre_octets/ftell(dst));

   return 0;
}

/*
Decompresse le fichier src
Le résultat se trouve dans dst
*/
char huffman_decompacter(FILE *src, FILE *dst, FILE *frq)
{
   int i, j, r;
   unsigned long nbre_octets;
   unsigned char nbre_ascii, bit_r;
   short index_frequence;
   short racine_arbre;
   unsigned char buffer[BUF_LEN];

   /* Lecture de la taille du fichier original */
   fread((char *)&nbre_octets, 4, 1, src);
   printf("Taille du fichier original : %d octets\n", nbre_octets);
   printf("Decompression en cours...\n");

   /* Lecture de la table des fréquences */
   if (frq==NULL)
   {
      index_frequence=huffman_lire_frequences(src);
   }
   else
   {
      fread(&nbre_ascii, 1, 1, src);
      index_frequence=huffman_lire_frequences(frq);
```

```
   }

   if (index_frequence==-1)
   {
      printf("Erreur de lecture de la table des frequences\n");
      return -1;
   }

   /* Construction de l'arbre à partir de la table des fréquences */
   racine_arbre=huffman_creer_arbre(index_frequence);

   /* Decompression du fichier source et écriture du résultat dans le fichier cible */
   j=racine_arbre;
   /* Lecture de BUF_LEN octets dans le fichier source */
   while ((r=fread(buffer, 1, BUF_LEN, src))>0)
   {
      /* Traitement octet par octet */
      for(i=0; i<r && nbre_octets>0; i++)
      {
         /* Traitement bit par bit */
         for(bit_r=0x80; bit_r!=0 && nbre_octets>0; bit_r>>=1)
         {
            if (buffer[i]&bit_r)
               j=arbre[j].branche1;
            else
               j=arbre[j].branche0;
            if ((arbre[j].branche0==-1) || (arbre[j].branche1==-1))
            {
               /*printf("%c", j);*/
               fputc((char)j, dst);
               nbre_octets--;
               j=racine_arbre;
            }
         }
      }
   }

   printf("Decompression terminee\n");

   return 0;
}

/*
Fonction principale
Ouvre les fichiers en lecture ou écriture et alloue un espace mémoire suffisant
pour les structures
*/
int main(int argc, char *argv[])
{
   FILE *src, *dst, *frq;

   /* Syntaxe incorrecte */
   if (argc<4)
   {
      printf("%s c/d/f source destination [frequence]\n", argv[0]);
      return -1;
   }
```

```
   /* Ouverture du fichier source en lecture */
   if ((src=fopen(argv[2], "rb"))==NULL)
   {
      perror("fopen");
      return -1;
   }

   /* Ouverture du fichier cible en écriture */
   if ((dst=fopen(argv[3], "wb"))==NULL)
   {
      perror("fopen");
      return -1;
   }

   /* Ouverture d'un éventuel fichier de liste de fréquences, en lecture */
   if (argc>4)
   {
      if ((frq=fopen(argv[4], "rb"))==NULL)
      {
         perror("fopen");
         return -1;
      }
   }
   else
      frq=NULL;

   /* Allocation mémoire pour les données diverses necessaires à la construction de
l'arbre */
   if ((arbre_d=(struct arbre_data *)malloc(512*sizeof(struct arbre_data)))==NULL)
   {
      perror("malloc");
      return -1;
   }

   /* Allocation d'une zone mémoire pour l'arbre */
   if ((arbre=(struct arbre *)malloc(512*sizeof(struct arbre)))==NULL)
   {
      free(arbre_d);
      perror("malloc");
      return -1;
   }

   /* Compression ou décompression ? */
   if (*argv[1]=='c')
      huffman_compacter(src, dst, frq);
   else if (*argv[1]=='d')
      huffman_decompacter(src, dst, frq);
   else if (*argv[1]=='f')
      huffman_creer_fichier_frequences(src, dst);

   /* Libération de la mémoire */
   free(arbre_d);
   free(arbre);

   /* Fermeture des fichiers */
   fclose(src);
```

```
   fclose(dst);
   if (frq!=NULL)
      fclose(frq);

   return 0;
}
```