Code de Huffman

Dans ce TD, on va s'intéresser à l'utilisation d'arbres, représentés de différentes manières, pour construire un code de Huffman.

1. **Code** — Parmi les ensembles ci-dessous, lesquels représentent un code ?
   a. $\{100, 0100, 010\}$
   b. $\{00, 01, 10, 11\}$
   c. $\{000, 001, 10, 1100, 1101, 111\}$
   d. $\{00, 11, 001, 010, 100\}$
   e. $\{00, 01, 10, 11, w\}$ pour un mot binaire quelconque w de longueur différente de 2.

2. **Codage** — On considère le texte (dont on a ôté les accents pour plus de simplicité) : `la belle et la bete`
   a. Quelle est la longueur (en bits) de ce texte, lorsqu'il est codé en ASCII ?
   b. Quelle est sa longueur si on code toute ses lettres par des mots binaires de longueur minimum ?
   c. Construire un arbre de Huffman associé au texte.
   d. Quelle est la longueur (en bits) de ce texte, lorsqu'il est codé en utilisant un code de Huffman ?
   e. Afin de construire tous le même arbre de Huffman, on choisit de placer toujours à gauche l'arbre de plus petit poids ou bien, en cas d'égalité, l'arbre qui contient la plus petite lettre (ceci n'a pas de raison d'être pour l'implantation).
   Donner la construction de cet arbres utilisant un tableau de nœuds.
   f. Comment transmettre le code de Huffman précédent ?
   g. Donner le contenu du fichier compressé.

3. **Décodage** — Expliquer comment décompresser le message suivant :

```
111001['e']0001['a']1['d']01['g']1['n']001['o']1['y']1['t']0001['i']1['p
']1['␣']001['c']1['m']01['r']1['s']11000011001000101001111011110111010011
000101110110011110001101100000111010111010011111111010000101000101010010
011001110011011001011100100 1110
```

4. **Performance**
   a. Quelles sont la hauteur minimale et la hauteur maximale pour un arbre de Huffman sur un alphabet $\mathcal{A}$ quelconque ? Et si $\mathcal{A}$ est l'ensemble des caractères ASCII ?
   b. Pour quelles distributions de lettres obtient-on de tels arbres ?

Dans les exercices qui suivront, on utilisera le type et les définitions C suivants :

```c
#define NBLETTRES 256
#define NBNOEUDS (NBLETTRES * 2 - 1)

typedef struct {
  unsigned char lettre; /* lettre représentée (si applicable) */
  int occur;            /* nombre d'occurrences */
  int fg;               /* indice du fils gauche (si applicable) */
  int fd;               /* indice du fils droit (si applicable) */
} NoeudHuffman;

typedef struct {
  int code[NBLETTRES];  /* code du caractère (sur NBLETTRES bits au plus) */
  int nombrebit;        /* longueur du code (en nombre de bits) */
} Codage;
```

5. **Occurrences** — Écrire une fonction `int Compter(File *entre, Noeud arb[])` qui remplit le tableau `int arb`, de taille `int NBLETTRES`, avec le nombre d'occurences de chacun des octets du fichier. La fonction renverra le nombre de caractères distincts contenus dans le fichier.

6. **Arbres de Huffman** — Écrire une fonction `int CreerArbre(NoeudHuffman arbre[], int nbFeuilles)` qui crée l'arbre de Huffman à partir du poids des `nbFeuilles` lettres contenues dans le tableau `arbre`. La fonction renvoie l'indice de la racine de l'arbre construit.

7. **Code d'un caractère** — Écrire une fonction `int CreerCode(Codage C[], NoeudHuffman arbre[], int racine)` qui, pour chaque caractère `x` présent dans l'arbre, initialise la case `C[x]` avec le code de Huffman de `x`. La racine de l'arbre est à l'indice `racine`. La fonction renvoie la taille modulo 8 du texte compressé.

8. **Code de l'arbre** — Écrire une fonction `void CodeArbre(FILE *sortie, NoeudHuffman arbre[], int racine)` qui écrit dans le fichier manipulé grace à `sortie` la suite décrivant l'arbre de Huffman. On pourra se contenter d'écrire chaque bit sur un caractère ASCII tout entier.

9. **Compression** — Écrire la fonction `main` qui reçoit une référence de fichier en argument et effectue la compression dans un fichier avec l'extension `.ZIP`.