

Codage de Huffman

Dans ce TP, on va étudier le codage de Huffman.

Dans toute la suite, on utilisera la structure suivante :

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

typedef struct {
    unsigned char lettre;
    int nbOccur;
    int filsGauche;
    int filsDroit;
} Noeud;

typedef struct {
    Noeud noeuds[NBNOEUDS];
    int nbFeuilles;
    int nbNoeuds;
    int curseurFeuille;
    int curseurInterne;
} Huffman;
```

0. **Sortir un papier et un crayon** — Si vous ne réussissez pas cet exercice, votre enseignant ne viendra pas vous aider en cas de problème lors d'une des questions qui suivent.
1. **Stockage des occurrences** — On souhaite compter les lettres apparaissant dans un fichier manipulé par un pointeur `FILE *in`. Pour ce faire, on commence par identifier tout entier $i < \text{NBLETTRES}$ au caractère dont le code ASCII étendu vaut i .

Écrire une fonction `void stockeOccurrences(FILE* in, int occur[])` qui remplit un tableau `occur` formé de `NBLETTRES` entiers, et tel que `occur[i]` soit égal au nombre d'occurrences du caractère i dans le fichier.

Par exemple, si on l'applique sur un fichier contenant le texte `abracadabrara`, le tableau `occur` renvoyé sera

	0	0		0	6	2	1	1	0	0	0	0	0	0	0	0	0	0	0	3	0			
<i>indice</i>	0	1	⋯	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	⋯

car les codes ASCII des caractères `'a'`, `'b'`, `'c'`, `'d'` et `'r'` sont 97, 98, 99, 100 et 114.

2. **Comptage des lettres** — Le tableau `occur` ayant été rempli grâce à la fonction précédente, écrire une fonction `int compteLettres(int occur[])` qui renvoie le nombre de caractères différents étant apparus dans le fichier.

Dans l'exemple précédent, cette fonction renverra donc l'entier 5.

3. **Arbre de Huffman : 1^{ère} partie** — Écrire une fonction `void initNoeuds(Noeud noeuds[], int occur[])` qui, à partir du tableau `occur` obtenu en question 1 et de l'entier `taille` obtenu en question 2, initialise le tableau de `Noeuds`. Celui-ci devra contenir `2*taille-1` `Noeuds`, et avoir la propriété suivante : si un caractère `c` a été trouvé `n` fois dans le fichier, l'une des `taille` premières cases du tableau `noeuds`

devra être un `Noeud` dont le champ `lettre` est égal à `c`, le champ `nbOccur` est égal à `n`, et les champs `fil gauche` et `fil droit` sont égaux à `-1`. Enfin, le contenu des `taille-1` dernières cases du tableau n'aura pour l'instant aucune importance. On ne se préoccupera pas des noeuds non-utilisés (cela nous évite de faire une allocation).

Si l'on poursuit l'exemple précédent, on pourra donc obtenir le tableau suivant (unique à l'ordre près de colonnes, et en notant `?` le contenu des cases qui n'ont aucune importance), où chaque `Noeud` est simplement représenté par quatre cases situées sur la même colonne :

<i>lettre</i>	'a'	'b'	'c'	'd'	'r'	?	?	?	?
<i>nbOccur</i>	6	2	1	1	3	?	?	?	?
<i>fil gauche</i>	-1	-1	-1	-1	-1	?	?	?	?
<i>fil droit</i>	-1	-1	-1	-1	-1	?	?	?	?
<i>indice</i>	0	1	2	3	4	5	6	7	8

4. **Arbre de Huffman : 2^{ème} partie** — Écrire une fonction `void triNoeuds(Noeud noeuds[], int taille)` qui, à partir du tableau `noeuds` obtenu en question 3 et de l'entier `taille` obtenu en question 2, trie les `taille` premiers `Noeud` contenus dans le tableau `noeuds` par ordre croissant en fonction de leur nombre d'occurrences. En cas d'égalité, on demande que le `Noeud` dont le caractère possède le plus petit code ASCII soit placé en premier. On pourra s'aider de la fonction `qsort` de la librairie standard C.

Si l'on poursuit l'exemple précédent, le tableau que l'on obtiendra sera alors

<i>lettre</i>	'c'	'd'	'b'	'r'	'a'	?	?	?	?
<i>nbOccur</i>	1	1	2	3	6	?	?	?	?
<i>fil gauche</i>	-1	-1	-1	-1	-1	?	?	?	?
<i>fil droit</i>	-1	-1	-1	-1	-1	?	?	?	?
<i>indice</i>	0	1	2	3	4	5	6	7	8

5. **Arbre de Huffman : 3^{ème} partie** — Écrire une fonction `Huffman initHuffman(int occur[])` qui renvoie un nouvel objet `Huffman`, dont on ne connaît pour l'instant que les feuilles, calculées à l'aide des fonctions précédemment définies. Cet objet servira à construire pas-à-pas un code de Huffman complet. Pour ce faire, à tout moment, on souhaite que les champs `nbFeuilles`, `nbNoeuds`, `curseurFeuille` et `curseurInterne` aient la signification suivante :

- notre arbre contient `nbFeuilles` feuilles et `nbNoeuds` nœuds (feuilles + nœuds internes) en tout ;
- les feuilles stockées en positions `0,...,curseurFeuille-1` du tableau ont déjà un parent dans l'arbre, et les feuilles en positions `curseurFeuille,...,nbLeaf-1` n'en ont pas ;
- les nœuds internes en positions `nbLeaf,...,curseurInterne-1` ont déjà un parent dans l'arbre, et les nœuds internes en positions `curseurInterne,...,nbNoeuds-1` n'en ont pas.

Avec quelles valeurs doit-on initialiser les champs `nbFeuilles`, `nbNoeuds`, `curseurFeuille` et `curseurInterne` ?

Dans l'exemple précédent, on aura `nbFeuilles = 5`, `nbNoeuds = 5`, `curseurFeuille = 0` et `curseurInterne = 5`.

6. **Arbre de Huffman : 4^{ème} partie** — Écrire une fonction `int plusPetitNoeudOrphelin(Huffman* huff)` qui renvoie, parmi les nœuds de l'arbre de Huffman `*huff`, l'indice du nœud sans parent dont le nombre d'occurrences est minimal. En cas d'égalité, on choisira le nœud stocké le plus à gauche possible dans le tableau `huff->noeuds`. Comme on s'apprête à donner un parent au nœud dont on renvoie l'indice, on demande également que cette fonction mette à jour les champs `curseurFeuille` et `curseurInterne` de `*huff`.

Dans l'exemple précédent, la fonction renverra 0 et modifiera la valeur du champ `curseurFeuille`, désormais égal à 1.

7. **Arbre de Huffman : 5^{ème} partie** — Écrire une fonction `void augmenteArbre(Huffman* huff)` qui, à partir d'un arbre de Huffman `*huff` partiellement rempli, sélectionne les deux nœuds sans parent dont le nombre d'occurrences est minimal (en cases en positions `i` et `j` de `huff->noeuds`, disons), puis crée un nouveau nœud interne qui sera leur nouveau parent. On n'oubliera pas de mettre à jour les champs `nbOccur`, `filsGauche` et `filsDroit` de ce nouveau nœud, ni les champs `curseurFeuille`, `curseurInterne` et `nbNoeuds` de `*huff`.
8. **Arbre de Huffman :** — Écrire une fonction `Huffman construitHuffman(FILE* in)` qui, à partir d'un fichier manipulé par un pointeur `FILE* in`, crée l'objet `Huffman` représentant l'arbre de Huffman associé à ce fichier.

Si l'on reprend l'exemple précédent, on obtiendra ainsi un `Huffman` dont le champ `noeuds` sera le tableau

<i>lettre</i>	'c'	'd'	'b'	'r'	'a'	?	?	?	?
<i>nbOccur</i>	1	1	2	3	6	2	4	7	13
<i>filsGauche</i>	-1	-1	-1	-1	-1	0	2	3	4
<i>filsDroit</i>	-1	-1	-1	-1	-1	1	5	6	7
<i>indice</i>	0	1	2	3	4	5	6	7	8

et les champs `nbFeuilles`, `nbNoeuds`, `curseurFeuille` et `curseurInterne` vaudront respectivement 5, 9, 5 et 8.

9. **Code de Huffman** — Étant donné l'objet `huff` renvoyé par la fonction précédente, on souhaite construire un tableau `traduc` formé de `NBLETTERS char*`, pour y stocker les mots du code de Huffman correspondant à l'arbre `huff`. Plus précisément, si un caractère `c`, de code ASCII `i`, apparaît dans le texte à compresser, alors `traduc[i]` devra être le mot code correspondant au caractère `c`; si le caractère `c` n'apparaît pas dans le texte à compresser, alors `traduc[i]` devra être égal à `code = NULL`.

Écrire une fonction `char** construitCode(Huffman huff)` qui crée le tableau `traduc`, le remplit de manière appropriée puis le renvoie.

Par exemple, pour l'arbre précédent, le tableau `traduc` sera

<i>taille</i>	0	0		0	1	3	4	4	0		0	2	0	
<i>code</i>	NULL	NULL		NULL	'0'	'110'	'1110'	'1111'	NULL		NULL	'10'	NULL	
<i>indice</i>	0	1	...	96	97	98	99	100	101	...	113	114	115	...

10. **Affichage du code** — Écrire une fonction `void afficheCode(char** traduc)` qui affiche chaque lettre ainsi que son mot de code associé.
11. **Compression de Huffman : 1^{ère} partie** — Rappeler comment générer l'en-tête et l'éventuel *padding* final d'un texte compressé grâce à l'algorithme de Huffman.

Dans l'exemple ci-dessus, notre en-tête consistera

- tout d'abord, en 3 bits b_2 , b_1 et b_0 , choisis de sorte que l'on utilisera $(4b_2 + 2b_1 + b_0)$ bits de *padding*;
- puis en la représentation de l'arbre de Huffman, ici donnée par la chaîne de caractères suivante :

0 1 0 1 1 0 0 0 0 1 0 1 0	1 1 1 0 0 1 0 0 1 0	1 1 0 0 0 1 0 0 1 0	1 0 1 1 0 0 0 1 0 0 1 0	1 0 1 1 0 0 0 1 1 1 0	1 0 1 1 0 0 1 0 0
code ASCII du	code ASCII du	code ASCII du	code ASCII du	code ASCII du	code ASCII du
caractère 'a'	caractère 'r'	caractère 'b'	caractère 'c'	caractère 'd'	

12. **Compression de Huffman : 2^{ème} partie** — Écrire une fonction `void codeHuffman(FILE *in, FILE *out)` qui compresse, grâce au code de Huffman, le contenu du fichier `in` et écrit le résultat obtenu dans `out`.

Le fichier de sortie contiendra, dans cet ordre :

- une représentation, sur 3 bits, de l'entier qui permet de compter le nombre de bits de rembourrage (ou *padding*) que l'on devra insérer à la fin du texte ;
- une représentation du code de Huffman lui-même, comme précisé à la question précédente ;
- notre texte compressé proprement dit ;
- les éventuels bits de *padding* présents à la toute fin, que l'on choisira tous égaux à 0.

Par exemple, l'encodage du texte `abracadabra` que l'on suit depuis le début sera

```
10101011000010101110010010110001001011000111011001000110100111001111011010000000
```

On pourra s'aider de la commande `xxd -bits <fichier>` pour afficher le contenu du fichier binaire produit.

Des fonctions pour la lecture et l'écriture de bits sont données dans le fichier `ecritlireBit.c` disponible sur **E-learning**.

13. **Compression automatique** — Écrire une fonction `int main(int argc, char *argv[])` qui reçoit un nom de fichier et compresse celui-ci dans un nouveau fichier obtenu en ajoutant `.HUF` au nom de fichier.
14. **Décompression automatique** — Écrire un programme de décompression d'un fichier de suffixe `.HUF`. Le nom du fichier décompressé sera obtenu en remplaçant `.HUF` par `.DEC`.
15. **Tests expérimentaux** — Tester le programme de compression sur le fichier `crime-et-chatiment`, puis de nouveau sur le fichier `crime-et-chatiment.HUF`. Que constate-t-on ?