IN 103 - TD 04

Énoncé

ENSTA ParisTech

1 Compression et codage de Huffman

Le codage de Huffman s'appuie sur le fait que dans les fichiers, certains caractères apparaissent plus souvent que d'autres. Par exemple en Français, le 'e' est plus fréquent que le 'w'.

De cette observation, l'idée est d'attribuer aux caractères des codes de longueur variable en fonction de leur fréquence d'apparition. Les caractères les plus fréquents recevront des codes courts ; les caractères plus rares des codes plus longs.

La compression consiste alors à remplacer les caractères du fichier d'origine par les codes leur correspondant, puis d'assembler les séquences de bits en octets qui seront finalement écrits sur disque.

Il est possible de se baser sur des tables de fréquences en fonction de la langue du fichier encoder, mais ceci est peu satisfaisant du fait que l'on n'encode pas que du texte et qu'il n'est pas forcément facile de déterminer ladite langue.

Dans le cas de fichiers sur disque, donc dont le contenu est entièrement disponible (pas comme dans le cas d'un flux continu) une meilleure solution existe : lire le fichier et construire cette table de fréquences. À la place d'une réelle fréquence, on préférera une mesure ≪inverse≫ : le nombre d'occurrences (ça évite des divisions et permet de n'utiliser que des entiers).

Q1 On souhaite écrire une fonction qui prenne en entrée un nom de fichier, construise puis retourne une table d'occurrences. Les octets étant sur 8 bits, la table comportera forcément . . . 256 entrées.

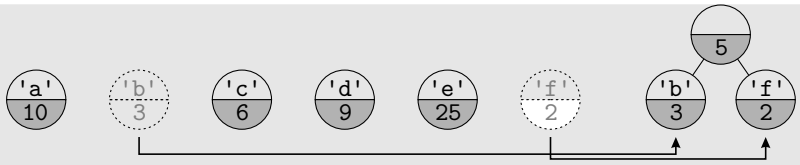
— Comment allez-vous allouer cette table ?

— Écrivez cette fonction.

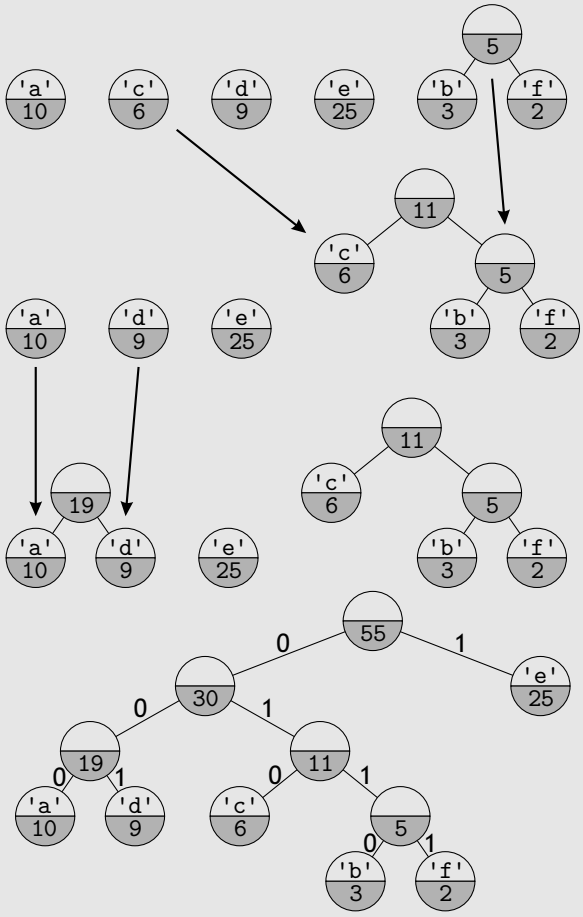
Pour lire le fichier caractère par caractère, vous pourrez au choix utiliser `scanf ()` ou la fonction `fgetc ()`. Cette dernière prend en argument le descripteur de fichier où lire et retourne un **int** contenant le code du caractère lu ou la valeur `EOF` pour signifier l'atteinte de fin de fichier.

Le calcul du code de Huffman de chaque lettre est réalisé en construisant un arbre. On commence par construire toutes les feuilles, c'est-à-dire une feuille par caractère de la table.

Puis, de manière itérative on choisit les deux arbres les plus rares (donc ayant une occurrence minimale) et on les fusionne en un nouvel arbre dont l'occurrence est la somme de celles de ses 2 fils.



On fusionne ainsi tous les arbres jusqu'à n'avoir plus qu'une racine. C'est l'arbre de Huffman.



Le code correspondant à un caractère correspond au parcours qu'il faut effectuer dans l'arbre pour atteindre la feuille contenant ce caractère : descendre à gauche correspond à un 0, descendre à droite à un 1. Dans l'arbre ci-dessus on obtient donc : `a=000`, `d=001`, `c=010`, `b=0110`, `f=0111`, `e=1`.

Q2 Écrivez la structure de données représentant un arbre de Huffman.

Q3 Écrivez une fonction `build_tree` qui construit l'arbre de Huffman pour un fichier dont le nom est donné en argument. Cette fonction se chargera de faire appel au calcul de fréquences que vous avez écrit en question **Q1**. Vous pouvez stocker tous les arbres que vous créez dans un tableau de 256 cases qu'il suffit de parcourir pour trouver l'arbre le plus rare.

Sous-questions pour vous guider :

1. Comment saurez-vous que vous avez terminé la fusion de tous les arbres ?
2. Comment, à chaque itération de fusion, allez-vous trouver les 2 arbres de poids minimaux ?

3. Comment allez-vous ≪boucher≫ le trou laissé par l'un des arbres fusionnés ?

Dans le fichier `code.(c|h)` vous est donnée la fonction :
char∗∗ get_codes (**struct** huff_node_t ∗tree)
qui prend en argument un arbre et retourne la table contenant, pour chaque caractère, sont code sous forme d'une chaîne de caractères '0' ou '1'. Cette table va servir à encoder le fichier initial.

Q4 Écrivez une fonction qui permet d'encoder un texte à l'aide du codage de Huffman. Cette fonction prendra en arguments le nom du fichier à encoder, un nom de fichier dans lequel écrire le résultat et la table des codes issue de l'analyse du premier fichier. Le codage se fait en lisant le fichier d'entrée caractère par caractère et en écrivant la séquence ≪binaire≫ associée dans le fichier de sortie (le fichier de sortie doit contenir des caractères `'0'` et `'1'`).

Dans la réalité, on n'écrit pas le code binaire sous forme de chaînes de caractères de `'0'` et de `'1'` : on travaille directement bit-à-bit sur des octets. Nous faisons donc une ≪simulation≫ de compression pour simplifier le programme en omettant les manipulations de bits.

Q5 Écrivez un `main` vous permettant de tester votre fonction. Pour vérifier que votre algorithme ≪compresse≫ effectivement, vous pourrez vérifier que le fichier codé a bien une taille plus petite que le fichier d'origine. Attention, dans le fichier codé chaque bit est écrit sur un caractère (donc un octet). En l'écrivant en binaire directement il serait 8 fois plus petit. Il faut donc vérifier que le fichier codé est moins de 8 fois plus grand que le fichier d'origine.

Q5-si-on-a-le-temps Expliquez en pseudo-code ou oralement l'algorithme de décodage.

1. Quel doit être le prototype de cette fonction ?
2. Quelles sont ses grandes étapes ?

Dans le fichier `decode.(c|h)` vous est donnée la fonction :
bool decode (**char** ∗in_fname, **char** ∗out_fname, **struct** huff_node_t ∗tree)
qui prend en argument un nom de fichier codé, un nom de fichier de sortie et un arbre de Huffman qui décode le contenu du premier fichier et écrit le résultat dans le second fichier.

Q6 Pour vérifier que votre algorithme de codage fonctionne correctement, décodez ce qu'il a codé. Si tout fonctionne, le fichier avant codage et celui issu du décodage doivent être exactement identiques.

Il vous suffit donc d'appeler la fonction de décodage `decode ()` donnée en lui transmettant l'arbre que vous avez construit pour encoder le fichier initial, ainsi que le nom du fichier codé.

Q7 Écrivez la fonction qui libère la mémoire utilisée par un arbre. Dans combien de cas est-elle nécessaire ? Utilisez-la où elle nécessaire.