Codage de Huffman

Mise en place

Pour débuter ce projet, nous avons créée une structure noeud afin de construire l'arbre binaire de recherche nécessaire à la mise en place du codage de Huffman. Cette structure comporte six champs :

- int val : ce champs est destiné au nombre d'occurence d'un caractère dans un texte
- char lettre : pour stocker ledit caractère
- int id : ce champs est destiné a identifié chaque noeud par un entier unique généré par la fonction indice(), qui se contente d'incrémenter une variable globale
- char* code : permettant de stocker le code binaire associé au caractère après le parcours de l'arbre
- struct noeud* fg, fd : les fils gauche et droit du noeud

La première partie du code est consacrée à la gestion de noeuds (création, suppression, affichage), des arbres (affichage, suppression, recherche de noeud), et enfin des listes (affichage, ajout d'éléments, suppression d'éléments, recherche d'éléments, suppression des doublons, taille de la liste), nécessaires à la gestion des noeuds dans certaines fonctions.

Création d'un arbre de Huffman

Grace aux fonctions précédemment évoquées, nous avons pu commencer le codage de Huffman.

Dans un premier temps, nous avons écris la fonction *mmaillon lecture().* Cette fonction crée une liste de noeuds, puis lis caractère par caractère le fichier *test.txt* (que l'on souhaite compresser). Pour chaque caractère, on vérifie s'il est déjà présent dans un noeud de la liste. Si oui on incrémente la valeur associée au noeud, sinon on ajoute un noeud à la liste avec pour valeur 1 et pour caractère le caractère que l'on est en train de lire. Cette fonction retourne la liste des noeuds associant à chaque caractère son nombre d'occurrences dans le fichier.

Les maillon de cette liste son ensuite fusionnés deux à deux dans la fonction *nnoeud fusion (mmaillon m).* On prend les deux maillons de la liste en paramètre dont la fréquence est la plus faible. Ils deviennent les fils gauche et droit d'un nouveau noeud dont la valeur est la somme des fréquences des deux noeuds sélectionnés. On ajoute ce noeud dans la liste et on en supprime les deux nous sélectionnés. On répète l'opération jusqu'à ce que la liste ne contienne plus qu'un seul noeud. Les fils étants placés selon leur valeur à droite ou à gauche lors de la fusion de deux noeud, le dernier noeud obtenu est un arbre binaire de recherche.

Code préfixe

Apres avoir créé notre arbre binaire de recherche repertoriant les caracteres et leur fréquence, nous eu besoin d'une fonction permettant d'associer à chaque caractère son code préfixe. Nous avons décidé de stocker ce code sous forme d'une chaîne de caractère dans le champs *code* de chaque noeud. La fonction *void affectation_code(nnoeud n, const char * prefix, char* car)* attribue à la racine le code 0. Elle parcours ensuite l'intégralité de l'arbre en ajoutant un 0 lorsqu'elle descend dans un fils gauche et un 1 lorsqu'elle descend dans un fils droit. Ainsi, à la fin du parcours d'arbre, chaque noeud est doté d'un code préfixe.

Compression

Une fois notre arbre binaire complet, avec ses codes préfixes, nous pouvons commencer la compression. La compression a lieu dans la fonction *void compression(nnoeud n)*. Cette fonction lis chaque caractère dans le fichier a compresser, et écris son code préfixe équivalent dans le fichier *compresse.txt*. On insère entre chaque code préfixé un espace pour faciliter la décompression. A la fin de cette fonction, on a un fichier *compresse.txt* contenant le text compressé précédé par la liste de tout les caractères du texte associé à leur fréquence. Cette liste est générée par la fonction *void aff_table_in_compression (FILE* fichier, nnoeud n)*. Cet index servira a reconstruire l'arbre à partir du fichier compressé lors de la décompression.
Pour faciliter la décompression, nous écrivons a travers cette fonction dans un autre fichier *decompresse2.txt*. Ce fichier ne contient que le texte compressé sans l'index.

Décompression

La décompression fonctionne de la même manière que la compression.
Avec la fonction *nnoeud create_arbre_from_compressed_file(char* string)*, on recréé l'arbre binaire de recherche associant à chaque caractère un code préfixe.
Ensuite, avec la fonction void *decompression (nnoeud n)*, on parcours le fichier compressé et pour chaque code préfixe rencontré, on écrit son caractère associé dans *decompresse.txt*. On obtient ainsi un fichier contenant le texte après décompression.

Utilisation

En ce qui concerne l'utilisation du code, au lancement, on peut mettre du texte en paramètre du lancement du programme, on peut mettre aussi le nom d'un fichier contenant du texte en paramètre du lancement du programme ou encore on peut choisir de laisser le fichier par défaut test.txt et changer ou non son contenu.
Ensuite automatiquement va se faire la compression à l'aide de l'arbre crée (car tout ça a été appelé dans le main), cette dernière va créer un fichier compression.txt et un fichier compression2.txt le premier contient de quoi recréer l'arbre et le résultat de la compression, le second quand à lui ne contient que le résultat de la compression.
Enfin la décompression se fait ensuite à l'aide du fichier compression2.txt, on retrouve le résultat de cette dernière dans le fichier décompresse.txt qui doit être au final le même que le fichier test.txt de départ.