Le principe du compresseur Huffman est de substituer des caractères présents dans un fichier par des suites de 0 et 1 en fonction de leurs occurrences. En principe un char est codé sur un octet. Lorsqu'un caractère a une forte récurrence, il convient de lui associer le code le plus court possible. Car si nous avons beaucoup de caractères il se peut certains codes dépassent les 8 bits initiaux. Mais le fait d'accorder un minimum de bits pour les caractères les plus récurrents va à la longue être en principe plus avantageux.

Par soucis de lisibilité, nous avons pris la liberté de séparer le projet en divers fichiers. En effet, les étapes précédant la compression étant relativement longues, nous avons trouvé judicieux de disposer tout cela de la sorte. Ainsi, on retrouvera séparément les signatures, l'écriture des fonctions et le programme principal. (Cela ne s'applique pas au programme de la décompression) Ce qu'on peut appeler comme étant la "compression finale" n'est possible qu'après être passé par diverses étapes.

La première consiste à déterminer les fréquences de chaque caractère afin de créer les nœuds adéquats. La seconde permet de construire l'arbre binaire selon l'algorithme de Huffman. Et la dernière nous octroie la possibilité de créer un code binaire unique relatif à chaque caractère présent dans le fichier (où aucun code n'est préfixe d'un autre pour éviter les ambiguïtés). Dès lors, il sera possible de faire la fonction compression.

Pour commencer, le comptage des caractères va parcourir au travers de la fonction fgetc le fichier source et récupérer chaque caractère. On crée ainsi un tableau dynamique qui va stocker le nombre de répétition des caractères. Il est bien-entendu nécessaire que les valeurs du tableau ne soient pas désallouées à la fin de la fonction. Il a été judicieux d'utiliser la valeur ASCII associé à chaque caractère comme indice du tableau. De cette manière lorsqu'on lit un caractère, on sait déjà sans avoir à faire d'opérations supplémentaires quelle case du tableau doit être modifiée. On remarquera la présence d'une constante nommée ASCII = 256. Nous l'avons créée car cette valeur revient assez fréquemment dans nos fonctions.

Il y a plusieurs fonctions d'affichage non essentielles qui permettent uniquement de répondre aux besoins de l'énoncé du projet.

Un fois les occurrences répertoriées. On peut procéder à la construction de l'arbre binaire. Nous avons choisi de créer notre arbre avec un tableau plutôt que des listes (même si cela est plus coûteux en mémoire) car étants moins à l'aise avec les listes, le tableau s'est révélé comme une solution évidente. Lorsqu'on construit un arbre pour chaque nœud, on doit avoir connaissance des "informations qui l'entoure". S'agissant d'un arbre binaire, il faut pouvoir connaître le nœud qui le précède, ceux qui le succède ainsi que son nombre d'occurrences. On peut aisément utiliser une structure (nœud) pour stocker tous ces attributs. L'arbre est donc un tableau où chaque case est un nœud. Il y en a exactement 511. On sait que s'il y a X feuilles au maximum et il y a X-1 nœuds maximums. Il peut donc y avoir au maximum 256 feuilles allant de 0 à 255 et les nœuds peuvent aller de 256 à 510. Il faut prendre soin de déclarer l'arbre en dehors des fonctions pour que sa portée ne soit pas limitée.

Cette dernière étape avant la compression que constitue la fonction parcoursArbre permet de stocker nos codes binaires en fonction de l'arbre construit. De la même manière que précédemment, on utilise un tableau global dans le fichier. Il contient des chaines de caractères. Pour se faire nous avons simplement besoin de l'arbre ainsi que de l'indice de la racine. La récursivité dans ce cas va être avantageuse. On parcourt d'abord les branches les plus à gauche en sortant progressivement des récursions on prend les branches de droites.

Jusque-là on a été guidé pour écrire notre code. C'est à partir de la compression que notre travail commence "réellement".

La compression va consister à inscrire le code binaire bit par bit dans un octet. À chaque octet rempli, nous le positionnons dans le fichier compressé. Pour parvenir à ce résultat, on a utilisé une variable de type char initialisé à 0000 0000. Cette variable doit être "vide" pour que nous puissions être en mesure de la remplir nous-même. Pour chaque caractère du fichier source lu, on écrit son code binaire bit par bit dans l'octet au moyen d'opérateurs binaires. Le ou "|" et le décalage à gauche "<<". De cette manière les codes binaires seront écrits dans le bon sens. Lorsque l'octet est plein, au travers de la fonction fputc, on peut le placer dans le fichier compressé.

Cependant, il va y avoir un problème, le fichier compressé ne contenant pas l'arbre permettant sa lecture. La décompression sera impossible. Il faut donc inscrire l'arbre dans le fichier compressé (à l'aide de fwrite) pour être en mesure de le déchiffrer en temps voulu. Et non pas dans un fichier tiers car cela nous obligerait à conserver ces deux fichiers toujours l'un avec l'autre. De plus dans ce cas on pourrait "perdre" l'index et ne plus être en mesure de procéder à la décompression. Toutefois, l'arbre étant d'une taille conséquente il va alourdir le fichier résultat. Nous reviendrons sur ce point par la suite. À la fin du fichier, on positionne un '\0' pour signifier la fin du code.

En dernier lieu, se trouve la fonction décompression. Dans un premier temps, nous avons pensé à utiliser le tableau de code binaire qui prend moins de place. Mais cela n'aurait pas fonctionné, il y aurait eu des ambiguïtés. Si on lit un 0 et qu'il y a plusieurs codes qui commencent par 0. Cela aurait été ambiguë, comment savoir lequel il fallait continuer à lire ? C'est pour cela qu'on a utilisé l'arbre. Après avoir fait de nombreux printf pour visualiser les valeurs numériques des octets du fichier compressé, nous avons remarqué que code était en complément à 2. Ainsi les valeurs oscillent entre [-127,127] et le premier bit détermine le signe du nombre. Donc pour savoir si le premier bit de l'octet est un 1 ou un 0 il suffit de tester si la valeur numérique de l'octet est positive ou négative. Positive : on lit un 0. Négative : on lit un 1. Ensuite, on fait un décalage à gauche et on se concentre sur le bit suivant. Dans la fonction parcoursArbre nous avons déterminé que les 0 sont à gauche et les 1 à droite. On fait cela jusqu'à arriver sur une feuille, on récupère son indice (valeur ASCII) ce qui nous donne le caractère correspondant et on passe au suivant. Cette fonction s'arrête dès lors qu'on a lu tous les codes binaires dont la quantité s'élève au nombre d'occurrence de la racine (total des caractères du fichier source).

À la fin, on se rend compte que la compression n'est vraiment intéressante que si le fichier source occupe un espace suffisamment grand. Car l'arbre que nous stockons dans le fichier résultat est relativement lourd. Ainsi si on a un petit fichier à compresser il sera plus volumineux une fois compressé. Dans ce cas la compression est alors inutile. Dans le cas contraire si on a un fichier d'une taille conséquente, le principe de compression étant efficace, le poids de l'index sera alors compensé. Et un gain positif se dégagera de cette compression.

Finalement par facilité d'exécution nous avons créé un makefile.