

# Projet IPF

## La commande pour tester

---

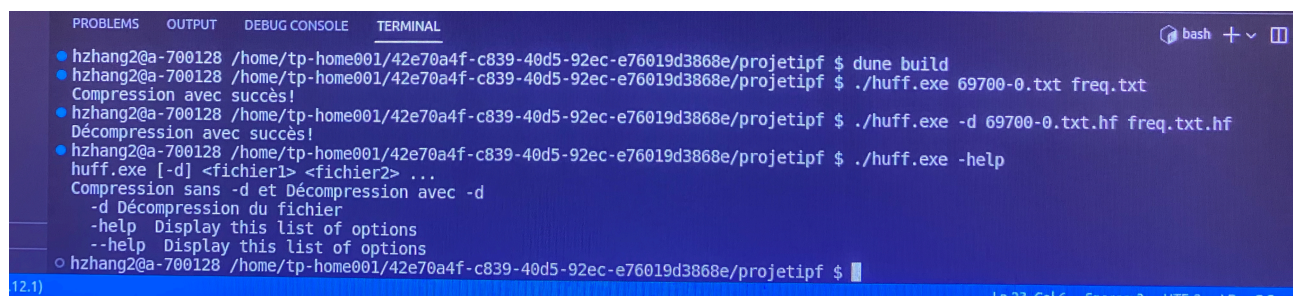
### Compress

`./huff.exe <fichier1> <fichier2> ...`

---

### Decompress

`./huff.exe -d <fichier1> <fichier2> ...`



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
• hzhang2@a-700128 /home/tp-home001/42e70a4f-c839-40d5-92ec-e76019d3868e/projetipf $ dune build
• hzhang2@a-700128 /home/tp-home001/42e70a4f-c839-40d5-92ec-e76019d3868e/projetipf $ ./huff.exe 69700-0.txt freq.txt
Compression avec succès!
• hzhang2@a-700128 /home/tp-home001/42e70a4f-c839-40d5-92ec-e76019d3868e/projetipf $ ./huff.exe -d 69700-0.txt.hf freq.txt.hf
Décompression avec succès!
• hzhang2@a-700128 /home/tp-home001/42e70a4f-c839-40d5-92ec-e76019d3868e/projetipf $ ./huff.exe -help
huff.exe [-d] <fichier1> <fichier2> ...
Compression sans -d et Décompression avec -d
-d Décompression du fichier
-help Display this list of options
--help Display this list of options
o hzhang2@a-700128 /home/tp-home001/42e70a4f-c839-40d5-92ec-e76019d3868e/projetipf $
```

## Partie I: Les types de données utilisés

---

### Tableau d'occurrence

Tableau d'occurrence : avec la fonction “Char\_freq”, nous pouvons facilement obtenir un tableau de longueur 256 contenant le nombre d'occurrences de chaque caractère dans tous les fichiers que nous voulons compresser. L'indice du tableau est le code ASCII correspondant au caractère et la valeur est le nombre d'occurrence.

---

### Liste de (int occurrence, arbre feuille)

Notre objectif est d'abord de générer un arbre de Huffman, mais avant cela, nous devons générer une nouvelle liste à partir de le tableau d'occurrence. Cette liste est (int \* arbre) liste et chaque élément est une paire constituée d'un arbre avec une seule feuille et le nombre d'occurrences du caractère (supérieur à 0). Lors de la création de notre liste, nous avons utilisé la fonction add pour obtenir une liste ordonnée, de la plus petite à la plus grande selon l'occurrence. Avec cette liste, nous pouvons générer notre arbre plus simplement.

---

### Arbre de Huffman

Cette structure est un arbre binaire c'est à dire que chaque nœud de l'arbre possède exactement deux fils (nœud interne) ou aucun (feuille). Le caractère correspondant à la feuille la plus proche de la racine de l'arbre est celui qui apparaît le plus souvent. En utilisant le 0 à gauche et le 1 à droite, nous pouvons déduire le nouvel encodage de chaque caractère et l'utiliser pour la compression.

---

## String array

Dans ce tableau de chaînes, nous stockons l'encodage de chaque caractère après compression. Ce tableau est converti à partir d'un arbre de Huffman. Nous avons implémenté cette conversion en utilisant la fonction de "code".

## Partie II: Fonction Compress

---

### Réalisation de la fonction

Au cours du processus de compression, nous devons lire le fichier original deux fois au total : la première fois pour construire la table de fréquence, et la seconde fois pour écrire les données compressées dans le nouveau fichier.

Après avoir obtenu la table de fréquence, on crée d'abord la liste (int occurrence, arbre feuille), puis on construit un arbre de Huffman à partir de cette liste, et enfin on construit un "dictionnaire", un tableau, à partir de l'arbre de Huffman, avec l'indice L'indice est le code ASCII correspondant au caractère et le contenu est l'encodage transformé à partir de l'arbre de Huffman.

L'étape suivante consiste à écrire le fichier. Afin de le décompresser, nous devons sauvegarder l'arbre de Huffman ainsi que les données. On enregistre donc d'abord l'arbre de Huffman, puis on lit tour à tour les caractères du fichier original, on trouve le code de Huffman correspondant au caractère selon le "dictionnaire" et on écrit le code dans le nouveau fichier, quand on a fini de lire on a aussi fini d'écrire. Pendant le processus de décompression, il suffit de reconstituer l'arbre de Huffman, puis de reconstituer le contenu à son tour.

---

### Sous fonction arbre

La fonction arbre est un élément extrêmement important de compress. En effet, toutes les compressions ultérieures sont basées sur les arbres de Huffman. Dans cette fonction, nous extrayons les deux plus petits éléments de la liste (ce sont exactement les deux premiers éléments de la liste) un par un, nous les fusionnons en un seul élément, l'occurrence devient la somme des deux occurrences, l'arbre des deux est utilisé comme sous-arbres gauche et droit pour construire un nouvel arbre, et cet élément est ensuite ajouté à la liste. Finalement, il ne reste qu'un seul élément dans la liste, et l'arbre dans cet élément est l'arbre de Huffman

que nous recherchons. De cette façon, nous pouvons obtenir que le caractère ayant le plus d'occurrences se trouve près de la racine de l'arbre binaire.

Au début, nous l'avons testé avec le mot satisfaisant sans aucun problème. Mais lorsque nous avons compressé un grand texte, le système n'a pas fonctionné. L'une des raisons en est notre fonction d'arbre.

```
let rec arbre h =  
  if is_singleton h then snd (List.hd h)  
  else  
    let x1 = fst (remove_min h) in  
    let x2 = fst (remove_min (snd (remove_min h))) in  
    let h0 = snd (remove_min (snd (remove_min h))) in  
    arbre (add ((fst x1) + (fst x2), Node((snd x1), (snd x2))) h0)
```

Vous pouvez voir que dans la fonction originale que nous avons utilisée, nous avons appelé la fonction “remove\_min” 5 fois, ce qui était complètement inutile, et en l'optimisant, nous avons obtenu la fonction finale qui n'a dû être parcourue que deux fois pour obtenir le même résultat

```
let rec arbre h =  
  if is_singleton h then snd(List.hd h)  
  else  
    let x1,h0 = remove_min h in  
    let x2,h1 = remove_min h0 in  
    arbre (add ((fst x1) + (fst x2),Node((snd x1), (snd x2))) h1)
```

---

## Sous fonction SauvegarderArbre

Le principal problème auquel nous sommes confrontés lors du stockage d'un Huffman est de s'assurer que l'arbre de Huffman est reconstitué exactement lorsqu'il est décompressé. Dans un premier temps, nous envisageons d'attribuer une valeur de 0 à chaque nœud, puis de préserver l'arbre à la fois pour la traversée préfixe et la traversée infix, puisque nous pouvons toujours reconstituer tout arbre binaire en fonction de ces deux dernières.

Cependant, l'inconvénient de cette méthode est que nous gaspillons beaucoup d'espace et qu'elle est inefficace car les arbres de Huffman ne stockent les données que sur les feuilles. Finalement, j'ai repris une idée de Stack Overflow : pour chaque feuille, on écrit le bit 1 et un octet du caractère correspondant ; pour chaque nœud, on écrit le bit 0, suivi de ses deux enfants de la même manière. Lors de la lecture d'un arbre de Huffman stocké de cette manière, lorsque le bit 1 est lu, un octet est lu et une feuille correspondante est créée ; lorsque le bit 0 est lu, les nœuds enfants gauche et droit sont décodés de la même manière et un nœud parent est créé sur la base des nœuds enfants gauche et droit. Voici le lien vers

la page: <https://stackoverflow.com/questions/759707/efficient-way-of-storing-huffman-tree>.

---

## Problèmes de sortie

La première difficulté que nous rencontrons lors de la sortie de caractères compressés est d'obtenir les caractères dans l'ordre original. Notre première idée est de relire le fichier original dans l'ordre et de construire une chaîne complète sur la base du dictionnaire que nous avons construit précédemment, qui est le codage de Huffman correspondant au fichier original, ensuite, nous lisons le contenu de cette chaîne un par un et l'écrivons dans le nouveau fichier. Comme la lecture et la sortie ne sont pas effectuées en une seule fois, lorsque le fichier est très grand, une traversée supplémentaire nécessitera une grande quantité de calculs et aussi un grand espace mémoire supplémentaire pour stocker les données converties en 0 et 1, de sorte que notre programme ne peut pas compresser de très grands fichiers. L'image ci-dessous est notre version originale.

```
let rec combine x s fo =
  try
    let n = input_byte fo in
    combine x (s^x.(n)) fo
  with End_of_file -> s
in
let contenu = combine x "" fo in
begin
sauvegarderArbre a os;
String.iter (fun x -> (write_bit os (int_of_char x - int_of_char '0'))) contenu;
finalize os
end
```

Lorsque nous avons regroupé ces deux étapes en une seule traversée, ce problème n'existe plus et nous avons réussi le test avec un livre. L'image ci-dessous montre notre version optimisée.

```
let rec loop0 x fo os =
  try
    let n = input_byte fo in
    String.iter (fun x -> (write_bit os (int_of_char x - int_of_char '0'))) x.(n);
    loop0 x fo os
  with End_of_file -> ()
in
begin
sauvegarderArbre a os;
loop0 x fo os;
finalize os
end
```

## Partie III : Les tests

---

### Test de arbre

Avant de tester le fichier compressé, nous devons tester l'arbre de Huffman que nous avons généré, car s'il y a un problème avec cette étape, il est impossible de la terminer plus tard. Notre fonction `testarbre` affiche le codage de Huffman de tous les caractères du terminal après compression. Nous avons trouvé des erreurs dans notre programme lors des premiers tests. Nous obtenons parfois des caractères avec le même préfixe, et cela est dû à un problème avec l'ordre de la gauche et de la droite dans le "code".

---

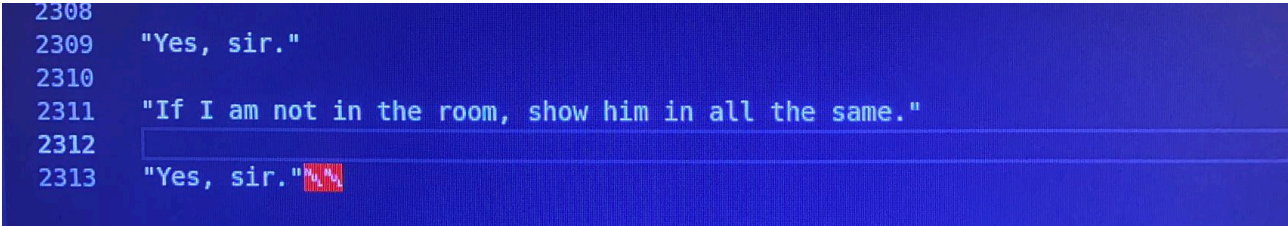
### Test décompresse

Nous avons d'abord testé avec des fichiers de plus petite taille, tels que Satisfaisant, et n'avons rencontré aucun problème avec aucun de nos programmes. Cependant, lorsque nous avons téléchargé une copie de Sherlock Holmes pour le tester, le programme n'a pas répondu. Comme déjà mentionné, nous avons optimisé le programme et obtenu des résultats satisfaisants. Le fichier compressé est maintenant de 63 kb au lieu de 108 kb, soit une réduction de plus d'un tiers par rapport au fichier précédent.

---

### Problème on a rencontré

Enfin, lors de la compression, nous n'avons aucun problème, mais lorsque nous décompressons le nouveau fichier txt, il y a toujours deux caractères supplémentaires à la fin. C'est ce que montre l'image ci-dessous.



```
2308
2309 "Yes, sir."
2310
2311 "If I am not in the room, show him in all the same."
2312
2313 "Yes, sir."■■■
```

Nous avons découvert que c'était parce que nous utilisons la fonction "finalize" à la fin de l'écriture dans la fonction "decompress" (Parce que dans "decompress" on écrit des octets au lieu de bits, alors il n'est pas besoin de l'utiliser), et lorsque cette fonction a été supprimée, nous avons pu obtenir un fichier qui était exactement le même que le texte original.

## Partie IV: Répartition

---

Tianwen GU

1. heap.ml
  2. char\_freq, lireArbre, sauvegraderArbre, code, arbre, compresse, decompress
  3. La ligne de commande
  4. Rapport
- 

Hongfei ZHANG

1. char\_freq, code, arbre, compress, decompress,
2. Test
3. Rapport