STRUCTURE DE DONNEES

# COMPRESSION ET DECOMPRESSION SELON L'ALGORITHME DE HUFFMAN

DORMOY Alexis, HABCHI Sonia, MAQSOOD Shahzeb

HABCHI Sonia

MAQSOOD Shahzeb

# Sommaire :



# Résumé

Ce rapport contient une introduction avec un rappel du problème énoncé, le plan du rapport, un rappel des conditions de développement, l'architecture de l'application ainsi que les principaux algorithmes détaillés et expliqués. Une mise au point du projet ainsi que les difficultés rencontrées et les solutions envisagés (et surtout retenu) pour gérer ces problèmes.
Il contient également une conclusion sur l'avancement et l'état actuel du projet ainsi que ses perspectives d'évolution.

# Introduction

## Problème traitée

Un fichier texte est composée de caractères, ces caractères sont codés sur un ensemble de bits. Le plus souvent on utilise la norme ASCII qui permet de coder 128 caractères. Chaque caractère est codé sur 1 octet (8bits) de 0000 0000(0) à 1111 1111(127) et donc chaque caractère prend la même place en mémoire à savoir 1 octet.
Nous cherchons donc à savoir s'il n'y a pas un moyen d'écrire du texte en prenant moins de place en mémoire et donc de compresser l'écriture des caractères en diminuant le nombre de bits nécessaire et donc réduire la taille en octet d'un fichier texte en le compressant. Il faut toutefois être capable de le décompresser par la suite, la compression doit donc être sans perte.

## Plan du document

Notre document prendra donc la forme suivante : un rappel du sujet qui correspond à la solution choisie pour la compression d'un fichier texte ainsi qu'une brève explication de son fonctionnement, de ses avantages et inconvénients. Puis de l'architecture de notre application pour voir l'articulation de l'application et les liens entre les fichiers ainsi que leurs utilités.
Une explication détaillée des principaux algorithmes pour comprendre leurs principes

Une explication du fonctionnement de l'application et des exemples d'utilisation.
Les différents problèmes rencontrés lors du développement, les différentes solutions envisagées et celle retenue.

Une conclusion sur l'avancement du projet et ses évolutions possibles.

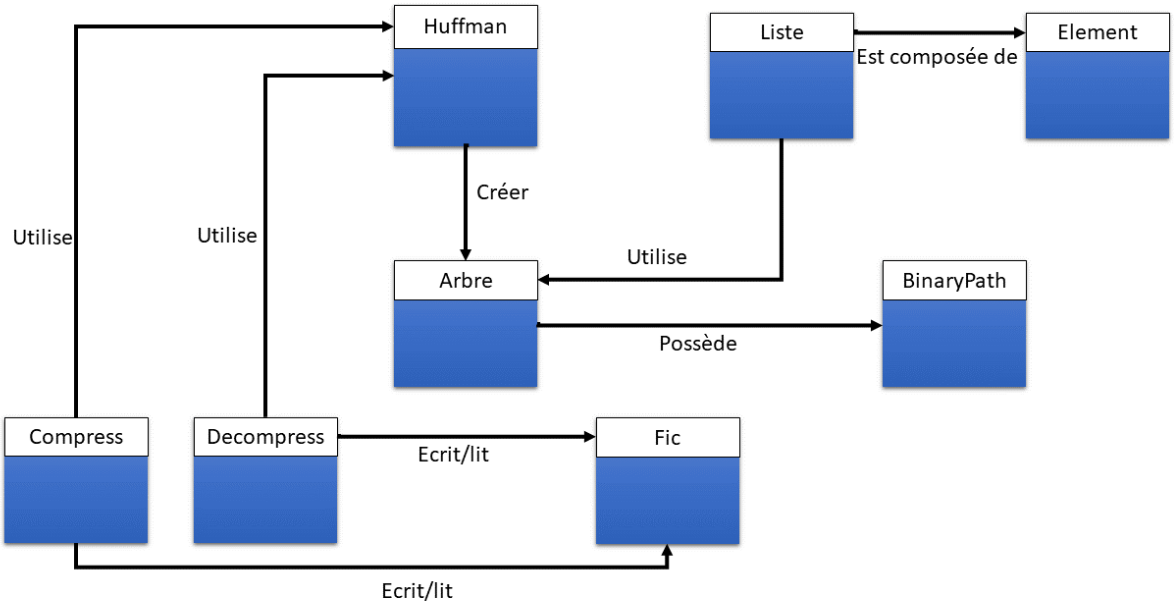
# Rappel succinct

Nous devons réaliser une application de compression/décompression de fichier texte en suivant l'algorithme de Huffman.
Cet algorithme est basé sur la création de l'arbre et de la table de Huffman ainsi que sur la lecture préalable du document avant sa compression. En effet, nous devons connaître les caractères présents dans le document ainsi que leur fréquence pour construire notre algorithme. Plus un caractère est fréquent, plus on l'associera a un code binaire court et à l'inverse un caractère peu présent sera associé à un code binaire plus long. L'avantage de cet algorithme est qu'il est « sans perte » et qu'il est relativement simple. Ces défauts sont qu'il faut connaître la table de Huffman correspondant au fichier pour pouvoir le décompresser, il faut donc écrire un entête dans le fichier compressé, cet entête ayant une certaine taille en octet, il n'est donc pas avantageux sur des fichiers très peu volumineux (-de 500octets, le fichier compressé sera plus lourd) et peu rentable sur des fichiers inférieurs à environ 1ko.

Il nous a été demandé de ne pas faire la distinction entre majuscule et minuscule (les majuscules deviendront donc des minuscules lors de la décompression).
Le fichier

# Architecture de l'application

## Diagramme des classes




## Explication

Pour créer l'arbre de Huffman, nous utilisons une liste chaînée d'élément. Chaque élément représente un arbre (au début des feuilles contenant un caractère et une fréquence). La structure BinaryPath permet de créer des « mots » binaire. Ces mots sont construits lors du parcours de l'arbre, ils servent à coder les caractères présents dans les feuilles sur un nombre de bits plus petit (principe de la compression) et servent aussi pour écrire le parcours infixe de l'arbre de Huffman dans l'entête du fichier compressé (0 on descend et 1 on monte). Ce sont les BinaryPath qui feront la table de Huffman.

Les fichier compress et decompress correspondent respectivement à la compression et décompression d'un fichier et utiliseront un fichier (fic) et l'arbre de Huffman pour fonctionner.
les fichiers Fic servent à l'ouverture, lecture et fermeture d'un fichier.

La table de Huffman est une liste chainée comprenant des elt_THuffman, ces éléments sont composés d'un caractère, d'un BinaryPath et d'un élément suivant

# Principaux algorithmes

## Création liste chaînée

Entrée : tableau d'entier de taille 128 (code ascii, l'indice = code ascii en décimal et la valeur la fréquence du caractère)

Sortie : liste chainée et triée des feuilles de l'arbre par ordre croissant selon la fréquence

```
Liste *l=NULL;
l=create_liste();
for(int i=0;i<128;i++)
{
    if(occurence[i]!=0)
        add_in_initial_liste(l,(char)i,occurence[i]);
}
```

```
//construction de la liste initial
//la liste est construite trié par ordre croissant
void add_in_initial_liste(Liste *l,char c,int f)
{
    Arbre *abr=create_feuille(c,f);
    add_in_liste(l,abr);
}
```

```
void add_in_liste(Liste *l,Arbre *abr)
{
    Element *e=l->first;
    if(e==NULL){
        Element *ajout=create_element(abr,NULL);
        l->first=ajout;
        return;
    }
    //ajout en debut
    if(abr->freq<=e->abr->freq)
    {
        Element *ajout=create_element(abr,e);
        l->first=ajout;
        return;
    }
    while(e->next!=NULL)
    {
        //Si l'element suivant a une plus grande frequence
        //Alors on l'ajoute ici
        if(abr->freq<e->next->abr->freq)
        {
            Element *ajout=create_element(abr,e->next);
            e->next=ajout;
            return;
        }

        e=e->next;
    }
    Element *ajout=create_element(abr,NULL);
    e->next=ajout;
    return;
}
```

## Création arbre Huffman

Entrée : Liste chainée des feuilles triées par ordre croissant de fréquence

Sortie : Liste dont le premier élément est la racine de l'arbre de Huffman

```
void construction_arbre_huffman(Liste *l)
{
    Element *e=l->first;
    //Tant qu'il y a au moins 2 elt dans la liste
    while(e->next!=NULL)
    {
        Arbre *abr=create_arbre(e->abr,e->next->abr);
        add_in_liste(l,abr);
        delete_elt(l,e->next);
        delete_elt(l,e);
        e=l->first;
    }
}
```

## Création table de Huffman

Entrée : Racine de l'arbre de Huffman, table de Huffman vide et un BinaryPath vide

Sortie : table de Huffman remplie

/ ! \ i est ici une variable globale qui permet de gérer l'indice de la table avec la récursivité

```
void creation_table_huffman(Arbre *abr,T_huffman *th,BinaryPath *bp)
{
    remplissage_table(abr,th,bp);
    i=0;
}
```

```c
void remplissage_table(Arbre *abr,T_huffman *th,BinaryPath *bp)
{
    if(abr==NULL)
        return;
    ajout_bits(bp,'0');
    remplissage_table(abr->gauche,th,bp);
    enlever_bits(bp);
    if(isFeuille(abr))
    {
        add_in_tab(th,abr->car,bp);
        i++;
    }
    ajout_bits(bp,'1');
    remplissage_table(abr->droite,th,bp);
    enlever_bits(bp);
}
```

## Compression du texte

Entrée : Fichier texte à compresser, Fichier où écrire le texte compressé, table de Huffman remplie

Sortie : Fichier de sortie compressé remplie

```c
void compresser_texte(FILE *entry,T_huffman *th,FILE *outfile)
{
    unsigned char buffer=0;
    int taille_buffer=0;
    rewind(entry);
    char c=fgetc(entry);
    while(c!=EOF)
    {
        ecrire_binary_path(c,th,outfile,&buffer,&taille_buffer);
        c=fgetc(entry);
    }
    //Ecrire ce qu'il reste dans le buffer et le vider
    buffer=(buffer<<(8-taille_buffer));
    fwrite(&buffer,1,1,outfile);
    //Il reste des 0 "en trop" placé a la fin du fichier
    //Mais ils pésent moins d'un octet -> on les laisse
}
```

```c
void ecrire_binary_path(char c,T_huffman *th,FILE *outfile,unsigned char *buffer,int *taille)
{
    //Gestion majuscule
    if((int)c>64 && (int)c<91)
        c=c+32;
    THuffman_elt *actuel = th->first;
    while(actuel!=NULL)
    {
        if(actuel->car==c)
        {
            for(int i=0;i<actuel->code.longueur;i++)
            {
                if(actuel->code.Bcode[i]=='1')
                    *buffer=(*buffer<<1) | 1;
                else
                    *buffer=(*buffer<<1) | 0;
                *taille=*taille+1;
                //L'octet est rempli, on vide le buffer
                if(*taille==8)
                {
                    fwrite(buffer,1,1,outfile);
                    *buffer=0;
                    *taille=0;
                }
            }
        }
        actuel=actuel->next;
    }
}
```

Ici, on écrit le BinaryPath de chaque caractère lu

Jusqu'à remplir le buffer (1octet), puis on vide le buffer et on réitère l'opération.

## Décompression du texte

Entrée : Table de Huffman remplie, le texte compressé en binaire, le nombre de caractère, les arguments passés en paramètre.
Sortie : texte décompressé dans la console de sortie (ou dans un fichier si l'option '-w' est demandé)

```
void traduction_bp(T_huffman *th,BinaryPath *binaire,char* nb,char **argv)
{
    FILE *out=NULL;
    if(recherche_argument(argv,"-w")==1)
    {
        char *fichier_sortie=get_file_name(argv);
        out=fopen(fichier_sortie,"w+b");
        if(out==NULL)
            exit(-32);
    }

    int objectif=atoi(nb);
    BinaryPath *mot=newBinaryPath();
    int lettre_trouve=0;
    int i=0;
    while(i<binaire->longueur)
    {
        ajout_bits(mot,binaire->Bcode[i]);
        if(lettre_trouve<objectif)
        {
            //ajout_bits(mot,binaire->Bcode[i]);
            if(recherche_mot(mot,th,out,argv))
            {
                lettre_trouve++;
                while(mot->longueur!=0)
                    enlever_bits(mot);
            }
        }
        i++;
    }
    if(!zero(mot))
    {
        BinaryPath *last=newBinaryPath();
        int i=0;
        while(!zero(mot))
        {
            ajout_bits(last,mot->Bcode[i]);
            if(recherche_mot(last,th,out,argv))
            {
                while(last->longueur!=0)
                    {enlever_bits(last);
                    enlever_premier_bits(mot);}
                    i=0;
            }
            else
                i++;
        }
    }
    fclose(out);
}
```

```
int recherche_mot(BinaryPath *mot,T_huffman *th,FILE *out,char **argv)
{
    THuffman_elt *actuel=th->first;
    while(actuel!=NULL)
    {
        if(egalite_bp(mot,&actuel->code))
        {
            printf("%c",actuel->car);
            if(recherche_argument(argv,"-w")==1)
                fprintf(out,"%c",actuel->car);
            return 1;
        }
        actuel=actuel->next;
    }
    return 0;
}
```

Ici, on ajoute à un binarypath chaque bit lu, on regarde s'il constitue un mot de la table de Huffman, dès qu'un mot est trouvé, on écrit le caractère correspondant et on vide le buffer (binarypath)

# Mise au point

## Explication

Le programme a été conçu de la manière suivante :
Réflexion en trinôme sur la façon de réaliser le projet, découpage du projet en plusieurs parties, distribution du travail, assemblage des travaux de chacun.

Nous avons codé le projet à l'aide de l'IDE Code Blocks sur des machines Windows avec MinGW

## Exemple d'utilisation

Compiler les fichiers :

gcc main.c compress.c decompress.c arbre.c liste.c element.c fic.c huffmain.c binarypath.c -o nom_programme

(Ici nous avons choisi out comme nom de programme)

Cette commande compile les fichiers .c et produit le fichier out.exe (sur Windows)

On tape ensuite la commande suivante : out help

```
C:\Users\adormoy2\Desktop\algo\Huffman_Project>gcc main.c compress.c decompress.c arbre.c liste.c element.c fic.c huffman.c binarypath.c -o out

C:\Users\adormoy2\Desktop\algo\Huffman_Project>out help
/////PAGE D'AIDE DU PROGRAMME DE COMPRESSION/DECOMPRESSION/////

Ce programme vous permet de compresser et de decompresser des fichiers textes

Liste des arguments du programme:

help :
        affiche l'aide
compress nom_fichier.txt :
        permet de compresser le fichier txt
decompress nom_fichier.hf :
        permet de decompresser un fichier compressÚ
both nom_fichier.txt :
        compresse ET decompresse un fichier /!\ Uniquement pour les tests dev/!\
-w fichier_sortie :
        mode ecriture, decompresse le fichier dans un .txt
-v fichier_sortie :
        mode verbose, affiche l'arbre de Huffman (uniquement pour compress)
-V fichier_sortie :
        mode verbose, affiche la table de Huffman
-e fichier_sortie :
        mode entete, affiche l'entete et les caracteristiques du fichier (uniquement pour decompress)
```

Compression d'un fichier avec affichage de l'arbre(-V) et de la table de Huffman(-v) (nous avons créé un fichier 'test.txt' contenant le texte « mohamed maachaoui »

Cela produit en sortie le fichier compressé test.txt.hf

```
C:\Users\adormoy2\Desktop\algo\Huffman_Project>out compress test.txt -v -V
lancement de la compression du fichier test.txt ....
TABLE DE HUFFMAN :
' ' -----> 1|1|0|1|
'a' -----> 0|0|
'c' -----> 1|1|0|0|
'd' -----> 1|0|1|1|
'e' -----> 1|0|1|0|
'h' -----> 0|1|0|
'i' -----> 1|0|0|1|
'm' -----> 1|1|1|
'o' -----> 0|1|1|
'u' -----> 1|0|0|0|
(17)
|___0___
        (8)
        |___0___
                (4) : a
        |___1___
                (4)
                |___0___
                        (2) : h
                |___1___
                        (2) : o
|___1___
        (9)
        |___0___
                (4)
                |___0___
                        (2)
                        |___0___
                                (1) : u
                        |___1___
                                (1) : i
                |___1___
                        (2)
                        |___0___
                                (1) : e
                        |___1___
                                (1) : d
                                (1) : d
        |___1___
                (5)
                |___0___
                        (2)
                        |___0___
                                (1) : c
                        |___1___
                                (1) :
                |___1___
                        (3) : m
Compression : OK !
```

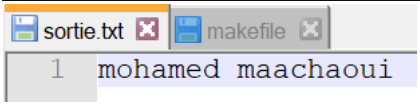
Décompression du fichier en mode écriture avec affichage de l'entête et de la table de Huffman :

Cela produit le fichier sortie.txt avec le texte «mohamed maachaoui »

```
C:\Users\adormoy2\Desktop\algo\Huffman_Project>out decompress test.txt.hf -w sortie.txt -v -e
lancement de la decompression du fichier test.txt.hf ....
nombre de caractere total : 17
nombre de caractere distinct : 10
liste des caracteres : ahouiedc m

chemin de l'arbre en parcours infixe :
||0|1|0|0|1|0|1|1|1|0|0|0|0|1|0|1|1|0|0|1|0|1|1|1|0|0|0|1|0|1|1|0|1|

TABLE DE HUFFMAN :
' ' -----> 1|1|0|1|
'a' -----> 0|0|
'c' -----> 1|1|0|0|
'd' -----> 1|0|1|1|
'e' -----> 1|0|1|0|
'h' -----> 0|1|0|
'i' -----> 1|0|0|1|
'm' -----> 1|1|1|
'o' -----> 0|1|1|
'u' -----> 1|0|0|0|
mohamed maachaoui
```




La chaîne « mohamed maachaoui » n'étant pas très grande, le fichier compressé est plus lourd à cause de l'entête, en revanche en essayant avec un texte type lorem_ipsum (vous trouverez le dans l'archive des sources) le texte pèse 2ko et le fichier compressé 1 ,19ko.

## Difficultés rencontrées et solution choisie

La plus grande difficulté rencontré a était de reconstruire la table de Huffman à partir de l'entête du fichier compressé, nous avons choisi de resimuler le parcours infixe à l'aide des 0 et 1 et en recréant les même binarypath que pour la compression lorsqu'on arrivait sur une feuille.
De plus sur notre éditeur de texte(NotePad++), les retours à la ligne sont composés d'un '\r' et d'un '\n' ce qui nous a causé de nombreux bug .
Un autre problème était de savoir comment être sûr d'écrire plusieurs caractère sur 1 seul octet, pour cela nous avons utilisé la fonction fwrite qui permet de préciser la taille en octet de ce que l'on écrit.

La gestion de l'entête nous a également causés des soucis. Car nous utilisons la fonction fgets() pour lire dans l'entête du fichier compressé, cette fonction lit une ligne entière , or sur la ligne des caractères présent dans le document, s'il y a un '\n' ou un '\r' par exemple, cela modifie le nombre de ligne sur lesquelles sont écrit les caractères.

La solution retenue a été de vérifier si après lecture de la ligne, la taille du tableau de caractère récupérer était égal au nombre de caractère distinct du fichier (représenté par la 2$^{e}$ ligne de l'entête)

Si ce n'est pas le cas alors on continue de lire la suivante et on fait une concaténation.

Une autre solution aurait été d'utiliser fread() en précisant la taille d'octet de données à lire.

Nous avons également eu un souci au niveau du temps, la partie de décompression a mis plus de temps que prévu à cause de certains bugs rencontrés

# Conclusion

## Etat d'avancement

Le projet est actuellement fonctionnel, nous n'avons pas encore passé le projet en mode release, il a donc été développé et testé uniquement en mode debug, ce qui réduit ses performances.
Il n'a pas non plus était testé sur des OS autres que Windows 10, nous ne pouvons donc pas garantir sa compatibilité avec un système Unix et avec un MinGW version 5.1.0, nous ne garantissons donc pas non plus son fonctionnement sur des versions antérieur. Il ne prend en compte que le fichier .txt et le fichier ne doit pas avoir d'espace dans son nom.

Certaines parties du code peuvent encore être factorisés / optimisés

## Perspectives d'évolution

Actuellement , comme le programme gère les arguments donnés, il ne prend pas en charge les fichiers contenant un espace dans leur noms (le programme considéra cela comme 2 arguments distincts)
Il faudrait aussi rajouter la possibilité d'afficher l'arbre de Huffman pour la décompression ainsi qu'un mode debug ou graphique pour montrer chaque étape.
Une autre évolution possible serait de prendre d'autres extension en compte, pour pouvoir par exemple compresser des images, des fichiers sonores etc etc…..

Réaliser une application graphique comme WinRar pourrait être une autre amélioration du projet.

Réaliser un MakeFile pour la compilation en ligne de commande permettrait de gagner du temps et éviterait les lignes de commandes trop grandes.

Augmenter le nombre de contrôle pour éviter les bugs et augmenter le nombre de code de retour pour mieux connaître d'où proviennent les erreurs (actuellement il n'y a que 5 code de retour possible pour le programme : 1 pour indiquer que la page d'aide a été demandé, 0 si le programme s'est bien passé , -1 si aucun argument n'a était saisi, -2 si mauvais arguments et -3 si problème lors d'une allocation de mémoire)
on pourrait ajouter des codes / vérification pour la libération mémoire, les opérations, les castes……