



Ministre de l'Enseignement Supérieur et de la Recherche

Université Sorbonne Paris Nord - Villetaneuse -

Institut Ingénieur Sup Galilée



Projet: Codage de Huffman

Module Programmation Avancée

Année universitaire 2024/2025

Date du devoir: 09 Février 2025

Nom: SEBA

Prénom: Mohammed Rabie

Identifiant: 12409891

Contents

1	Introduction	4
2	Algorithme de Huffman	4
2.1	Principe de base	4
2.2	Structures de données	4
2.3	Exemple de construction de l'arbre de Huffman	5
3	Implémentation	6
3.1	Approche choisie	6
3.2	Structure des données	6
3.3	Fonctions principales	7
3.3.1	Création d'un nœud	7
3.3.2	Insertion dans le tableau	7
3.3.3	Extraction du minimum	7
3.3.4	Construction de l'arbre de Huffman	8
3.3.5	Compression du fichier	8
3.4	Compression du texte	9
3.4.1	Exemple de fichier compressé	9
3.4.2	Interprétation du fichier compressé	9
4	Conclusion	10
4.1	Analyse de la complexité	10
4.2	Limites et perspectives	10
4.3	Exemple de compression	11
5	Instructions pour l'utilisation	11
5.1	Compiler le programme	11
5.2	Exécuter le programme	11
5.3	Résultat attendu	11
6	Résumé sur les détails du programme	12
6.1	Structures de données	12
6.2	Fonctions principales	12
7	Exemple de fichiers	12
8	Remarques supplémentaires	12
9	Principe de décompression	13
9.1	Détails sur la construction de l'arbre de Huffman	13
9.1.1	Format de stockage dans le fichier compressé	13
9.1.2	Parcours du fichier et reconstruction de l'arbre	13
9.1.3	Exemple de code pour la reconstruction de l'arbre	14
9.2	Construction de l'arbre de Huffman pour la décompression	16
9.2.1	Complexité de la construction de l'arbre	17
9.3	Décodage du fichier compressé	18
9.4	Libération de l'arbre de Huffman	19
9.5	Résumé sur les détails du programme	19

9.5.1	Structures de données	19
9.5.2	Fonctions principales	19
10	Instructions pour l'utilisation	20
10.1	Compiler le programme	20
10.2	Exécuter le programme	20
10.3	Résultat attendu	20

1 Introduction

Le codage de Huffman est une méthode de compression de données sans perte, largement utilisée pour réduire la taille des fichiers tout en conservant l'intégrité des informations. Cet algorithme attribue des codes binaires de longueur variable aux symboles en fonction de leur fréquence d'apparition, les caractères les plus fréquents recevant des codes plus courts tandis que les caractères rares se voient attribuer des codes plus longs. Cette approche permet ainsi d'optimiser l'espace de stockage et de minimiser le volume des données à transmettre, ce qui est particulièrement avantageux dans des domaines tels que la transmission de fichiers, le stockage de données et la compression multimédia.

Dans le cadre de ce projet, nous avons pour objectif d'implémenter en langage C un programme capable d'effectuer la compression et la décompression de fichiers en utilisant l'algorithme de Huffman. Pour cela, nous développerons une structure de données permettant de représenter l'arbre de Huffman, ainsi que les fonctions nécessaires à la génération des codes, à l'encodage et au décodage des fichiers.

Nous testerons notre implémentation sur différents types de fichiers, notamment des fichiers texte et des images aux formats JPG et BMP. Une analyse des performances sera réalisée afin d'évaluer l'efficacité de notre algorithme en termes de taux de compression et de temps d'exécution. Enfin, nous examinerons certaines variantes et optimisations possibles de l'algorithme pour améliorer encore davantage ses performances et son applicabilité à divers contextes.

2 Algorithme de Huffman

2.1 Principe de base

L'algorithme de Huffman repose sur la construction d'un arbre binaire optimal, appelé arbre de Huffman, qui permet de générer des codes binaires pour chaque symbole. Les étapes principales sont les suivantes :

1. Calcul des fréquences des symboles dans le fichier.
2. Construction de l'arbre de Huffman en fusionnant les nœuds de plus faible fréquence.
3. Génération des codes binaires en parcourant l'arbre.
4. Compression du fichier en remplaçant chaque symbole par son code binaire.
5. Décompression en utilisant l'arbre de Huffman pour reconstruire le fichier original.

2.2 Structures de données

Pour implémenter l'algorithme, nous utilisons les structures de données suivantes :

- **Noeud** : Représente un nœud de l'arbre de Huffman.
- **Tableau de Noeud** : Utilisée pour stocker les nœuds dans l'ordre croissant de fréquence.

2.3 Exemple de construction de l'arbre de Huffman

Voici un exemple illustrant la construction de l'arbre de Huffman pour la phrase "je regarde le paysage".

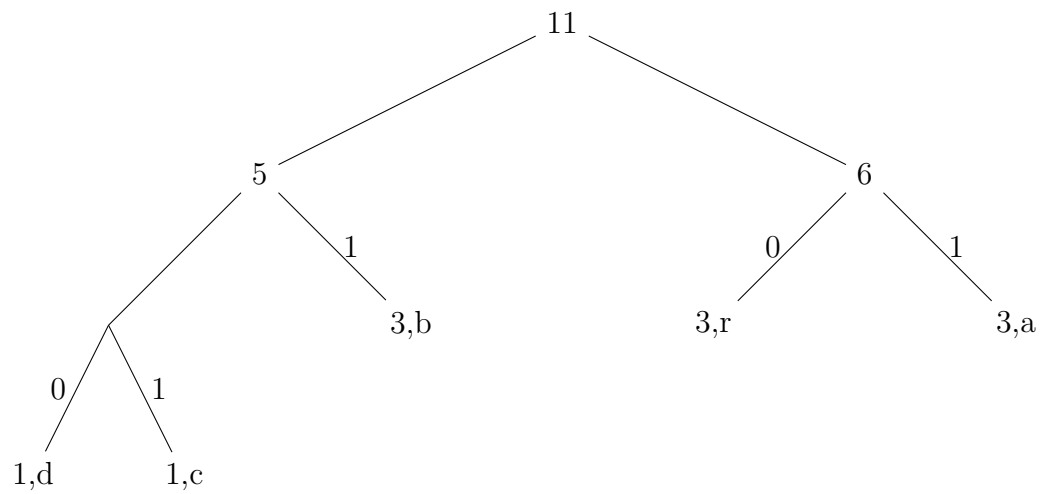


Figure 1: Arbre de Huffman pour le mot "abracadabra".

3 Implémentation

3.1 Approche choisie

L'implémentation de l'algorithme de Huffman a été réalisée en langage C, en suivant une approche modulaire. Voici les principaux choix effectués :

- **Structure de nœud** : Chaque nœud contient un caractère, sa fréquence, et des pointeurs vers les fils gauche et droit.
- **Tableau dynamique pour la file de priorité** : Un tableau dynamique a été utilisé pour stocker les nœuds, triés par fréquence croissante.
- **Gestion des bits** : Une méthode a été implémentée pour écrire et lire des bits dans un fichier.
- **En-tête du fichier compressé** : Les codes de Huffman sont stockés dans l'en-tête du fichier compressé, ce qui permet leur récupération lors de la décompression.

3.2 Structure des données

Les structures principales utilisées dans l'implémentation sont les suivantes :

- **Node** : La structure représentant un nœud de l'arbre de Huffman. Elle contient le caractère, la fréquence de ce caractère, un indicateur indiquant si le nœud est une feuille ou un nœud interne, ainsi que des pointeurs vers les nœuds enfants gauche et droit.
- **tab** : Une structure permettant de stocker un tableau dynamique de nœuds. Cette structure facilite la gestion de la file de priorité.

La structure Node est définie comme suit :

```
1 typedef struct node {
2     char data;           // Caractère stocké dans le nœud
3     uint64_t frq;        // Fréquence du caractère
4     bool useData;        // Indicateur si le champ data est
                           // utilisé
5     struct node *left;   // Pointeur vers le fils gauche
6     struct node *right;  // Pointeur vers le fils droit
7 } Node;
```

3.3 Fonctions principales

Le programme implémente plusieurs fonctions pour gérer la compression avec l'algorithme de Huffman.

3.3.1 Création d'un nœud

La fonction `createNode` permet de créer un nouveau nœud à partir d'un caractère, d'une fréquence et d'un indicateur indiquant si le nœud est une feuille.

```
1 Node* createNode(char data, uint64_t frq, bool useData) {
2     Node* newNode = (Node*)malloc(sizeof(Node));
3     newNode->data = data;
4     newNode->frq = frq;
5     newNode->useData = useData;
6     newNode->left = newNode->right = NULL;
7     return newNode;
8 }
```

3.3.2 Insertion dans le tableau

La fonction `insertInOrder` insère un nœud dans le tableau en maintenant l'ordre croissant des fréquences des caractères.

```
1 void insertInOrder(tab* t, Node* node) {
2     uint64_t i = t->size;
3     while (i > 0 && t->array[i - 1]->frq > node->frq) {
4         t->array[i] = t->array[i - 1];
5         i--;
6     }
7     t->array[i] = node;
8     t->size++;
9 }
```

3.3.3 Extraction du minimum

La fonction `extractMin` extrait le nœud ayant la plus petite fréquence de la file de priorité.

```
1 Node* extractMin(tab* t) {
2     if (t->size == 0) return NULL;
3     Node* minNode = t->array[0];
4     for (uint64_t i = 1; i < t->size; i++) {
5         t->array[i - 1] = t->array[i];
6     }
7     t->size--;
8     return minNode;
9 }
```

3.3.4 Construction de l'arbre de Huffman

La fonction `buildHuffmanTree` construit l'arbre de Huffman en combinant les deux nœuds ayant les plus petites fréquences à chaque itération.

```
1 Node* buildHuffmanTree(tab* t) {
2     while (t->size > 1) {
3         Node* left = extractMin(t);
4         Node* right = extractMin(t);
5         Node* parent = createNode('\0', left->frq + right->frq,
6                                     false);
7
8         if (left->frq < right->frq) {
9             parent->left = left;
10            parent->right = right;
11        } else {
12            parent->left = right;
13            parent->right = left;
14        }
15        insertInOrder(t, parent);
16    }
17    return t->array[0];
18 }
```

3.3.5 Compression du fichier

La fonction `compressFile` coordonne l'ensemble du processus de compression : elle compte les fréquences des caractères, construit l'arbre de Huffman, génère les codes binaires correspondants et écrit le fichier compressé.

```
1 void compressFile(const char* inputFilename, const char*
2     outputFilename, Node* root) {
3     FILE* inputFile = fopen(inputFilename, "rb");
4     FILE* outputFile = fopen(outputFilename, "wb");
5
6     if (!inputFile || !outputFile) {
7         perror("Erreur d'ouverture du fichier");
8         exit(EXIT_FAILURE);
9     }
10
11     char codes[MAX_CHAR][MAX_CHAR] = {0};
12     char code[MAX_CHAR];
13     generateCodes(root, code, 0, codes);
14
15     writeHeader(outputFile, codes);
16     writeCompressedText(inputFile, outputFile, codes);
17
18     fclose(inputFile);
19     fclose(outputFile);
20 }
```


3.4 Compression du texte

Le programme lit le fichier d'entrée et génère les codes de Huffman pour chaque caractère à l'aide de l'arbre de Huffman construit. Les codes sont ensuite écrits dans l'en-tête du fichier compressé, suivis des données compressées. Le fichier compressé contient les codes des caractères et la représentation binaire compressée du texte d'origine.

3.4.1 Exemple de fichier compressé

Voici un exemple de fichier compressé résultant de l'application de l'algorithme de Huffman :

```
a:0
b:111
c:101
d:100
r:110
----
01111100100010101111100
```

3.4.2 Interprétation du fichier compressé

Dans cet exemple, la première partie du fichier représente l'en-tête contenant les codes de Huffman attribués à chaque caractère présent dans le texte original. Chaque ligne associe un caractère à une séquence binaire unique, permettant ainsi de le reconstruire lors de la décompression.

Caractère	Code Huffman
a	0
b	111
c	101
d	100
r	110

Table 1: Table des codes de Huffman

La seconde partie, située après la ligne de séparation (----), contient la séquence binaire résultante de la transformation du texte d'origine en utilisant les codes de Huffman définis précédemment.

Lors de la décompression, cette séquence binaire est parcourue bit par bit et comparée aux codes enregistrés dans l'en-tête. Dès qu'une séquence binaire correspond à un code existant, le caractère correspondant est ajouté au texte décompressé. Ce processus se poursuit jusqu'à la fin de la séquence binaire, reconstituant ainsi fidèlement le fichier original, et c'est que ce qu'on va voir dans peu de temps

4 Conclusion

L'algorithme de Huffman est une méthode efficace de compression de données qui exploite la fréquence d'apparition des caractères pour attribuer des codes binaires de longueur variable, minimisant ainsi la taille du fichier compressé. Cette approche garantit une compression optimale sans perte, rendant le stockage et la transmission des données plus efficaces.

La construction de l'arbre de Huffman repose sur une file de priorité où les nœuds sont extraits et fusionnés pour générer un arbre binaire optimisé. L'encodage final est obtenu en parcourant l'arbre, générant des codes courts pour les caractères fréquents et des codes plus longs pour les caractères rares. (C'est que ce qu'on a parlé à l'introduction, et on venait de le confirmer ...)

4.1 Analyse de la complexité

L'algorithme de Huffman suit les étapes suivantes :

- **Calcul des fréquences** : Cette étape consiste à parcourir le fichier d'entrée une seule fois pour déterminer la fréquence de chaque caractère, ce qui prend un temps de $\mathcal{O}(n)$, où n est la taille du texte.
- **Construction de la file de priorité** : Chaque caractère unique est inséré dans une file de priorité (souvent implémentée sous forme de tas binaire). La construction d'un tas de k éléments a une complexité de $\mathcal{O}(k \log k)$, où k est le nombre de caractères distincts.
- **Construction de l'arbre de Huffman** : À chaque itération, les deux nœuds de plus petite fréquence sont extraits et fusionnés en un nouveau nœud, nécessitant un total de $k - 1$ fusions. L'extraction et l'insertion dans un tas de priorité ayant une complexité de $\mathcal{O}(\log k)$, la construction complète de l'arbre prend $\mathcal{O}(k \log k)$.
- **Génération des codes de Huffman** : Un parcours de l'arbre permet d'attribuer un code binaire à chaque caractère, ce qui se fait en $\mathcal{O}(k)$.
- **Encodage du texte** : La compression consiste à remplacer chaque caractère du fichier original par son code Huffman, ce qui prend $\mathcal{O}(n)$.

Ainsi, la complexité globale de l'algorithme est dominée par la construction de l'arbre de Huffman, soit $\mathcal{O}(k \log k)$, et le parcours du texte d'entrée, soit $\mathcal{O}(n)$. En supposant que $k \ll n$ dans la majorité des cas (puisque $k \leq 256$ pour du texte ASCII), la complexité peut être approximée à $\mathcal{O}(n + k \log k)$, ce qui reste très efficace pour des fichiers de grande taille.

4.2 Limites et perspectives

Bien que performant, l'algorithme de Huffman n'est pas toujours optimal. Il ne s'adapte pas aux changements de fréquence dans un flux de données et peut être inefficace sur certains types de fichiers où les répétitions d'un caractère sont mieux exploitées par d'autres techniques de compression.

4.3 Exemple de compression

Nous avons appliqué l'algorithme de Huffman sur un fichier texte de taille 902.5 KiB. Après compression, sa taille a été réduite à 537.2 KiB, ce qui représente un gain significatif en espace de stockage.



Figure 2: Comparaison des tailles des fichiers avant et après compression.

5 Instructions pour l'utilisation

5.1 Compiler le programme

Utilisez un compilateur C pour compiler le programme. Par exemple, avec GCC, vous pouvez exécuter la commande suivante :

```
gcc -o huffman_compress huffman_compress.c
```

5.2 Exécuter le programme

Le programme prend en argument le nom du fichier à compresser et le nom du fichier de sortie pour stocker le fichier compressé. Vous devez fournir ces deux arguments lors de l'exécution.

La syntaxe de la commande est la suivante :

```
./huffman_compress <nom_du_fichier_input> <nom_du_fichier_output>
```

Exemple :

```
./huffman_compress fichier.txt fichier_compressé.bin
```

5.3 Résultat attendu

Une fois l'exécution terminée, le programme compressé le fichier d'entrée et sauvegarde le fichier compressé dans le fichier de sortie spécifié. Un message de confirmation s'affiche :

Fichier compressé avec succès dans : fichier_compressé.bin

6 Résumé sur les détails du programme

6.1 Structures de données

- **Node** : Représente un nœud de l'arbre de Huffman. Chaque nœud contient un caractère, sa fréquence d'apparition dans le fichier, et des pointeurs vers ses fils gauche et droit.
- **tab** : Structure qui contient un tableau de nœuds représentant les éléments à insérer dans l'arbre de Huffman.

6.2 Fonctions principales

- **createNode** : Crée un nœud avec un caractère donné, sa fréquence et un indicateur si le caractère est utilisé.
- **createTab** : Crée et initialise un tableau de nœuds.
- **insertInOrder** : Insère un nœud dans le tableau en maintenant un ordre croissant de fréquence.
- **extractMin** : Extrait le nœud avec la plus petite fréquence.
- **buildHuffmanTree** : Construit l'arbre de Huffman en extrayant les nœuds de plus faible fréquence et en les combinant en un arbre binaire.
- **countFrequencies** : Compte la fréquence des caractères dans le fichier d'entrée.
- **generateCodes** : Génère les codes binaires en parcourant l'arbre de Huffman.
- **writeHeader** : Écrit l'en-tête du fichier compressé (codes des caractères).
- **writeCompressedText** : Écrit les données compressées dans le fichier de sortie.
- **compressFile** : Fonction principale pour compresser un fichier en appelant toutes les fonctions nécessaires.

7 Exemple de fichiers

- **Fichier d'entrée** : Le fichier que vous souhaitez compresser (par exemple `fichier.txt`).
- **Fichier de sortie** : Le fichier où le contenu compressé sera écrit (par exemple `fichier_compressé.bin`).

8 Remarques supplémentaires

- **Format de sortie** : Le fichier de sortie contient d'abord un en-tête qui associe chaque caractère aux codes binaires générés par l'algorithme de Huffman. Ensuite, le texte compressé est écrit avec les codes binaires correspondants aux caractères.
- **Optimisation mémoire** : Le programme utilise des structures comme `uint64_t` pour gérer les données volumineuses et garantir une bonne gestion de la mémoire.

9 Principe de décompression

Le processus de décompression d'un fichier compressé avec l'algorithme de Huffman consiste à reconstruire l'arbre de Huffman utilisé lors de la compression, puis à parcourir cet arbre à l'aide des bits présents dans le fichier compressé pour retrouver les caractères originaux.

9.1 Détails sur la construction de l'arbre de Huffman

La construction de l'arbre de Huffman est une étape essentielle dans le processus de décompression, car cet arbre contient les informations nécessaires pour décoder le fichier compressé. L'arbre de Huffman est utilisé pour associer chaque caractère à un code binaire unique, et il est construit en fonction de la fréquence d'apparition des caractères dans le fichier. Dans cette section, nous détaillons le travail effectué pour reconstruire l'arbre de Huffman à partir des données contenues dans le fichier compressé.

9.1.1 Format de stockage dans le fichier compressé

Le fichier compressé contient deux types d'informations importantes :

- **Les codes de Huffman**, qui sont des séquences binaires représentant chaque caractère.
- **Les caractères associés**, qui sont les symboles originaux utilisés dans le fichier compressé.

Ces informations sont stockées dans un format spécifique. D'abord, les codes de Huffman et les caractères associés sont enregistrés dans le fichier dans un format texte. Une séquence spéciale, telle que '—', marque la fin de cette section du fichier.

9.1.2 Parcours du fichier et reconstruction de l'arbre

La fonction `chargerHuffmanTree` parcourt le fichier compressé pour reconstruire l'arbre de Huffman. Cette fonction fonctionne selon les étapes suivantes :

1. **Lecture des caractères et des codes** : La fonction lit le fichier compressé caractère par caractère. Lorsqu'un caractère est trouvé, il est stocké dans une variable. Ensuite, le code binaire associé à ce caractère est lu.
2. **Construction des nœuds de l'arbre** : Chaque fois qu'un caractère et son code binaire associé sont lus, un nouveau nœud est créé. Ce nœud contient le caractère ainsi que le code binaire. Ce nœud est ensuite inséré dans l'arbre de Huffman en suivant les règles de construction de cet arbre : les caractères les plus fréquents sont plus proches de la racine, ce qui minimise la longueur de leur code binaire.
3. **Insertion dans l'arbre** : Le nœud nouvellement créé est inséré dans l'arbre de Huffman. L'arbre est parcouru de manière récursive pour trouver l'endroit approprié où insérer ce nœud.
4. **Fin de la lecture** : Lorsque la séquence '—' est rencontrée dans le fichier, cela indique que l'arbre de Huffman a été complètement chargé et qu'il est prêt à être utilisé pour la décompression.

9.1.3 Exemple de code pour la reconstruction de l'arbre

Voici un extrait de la fonction `chargerHuffmanTree` qui montre comment les codes sont lus et utilisés pour construire l'arbre de Huffman :

```
1 Node* chargerHuffmanTree(FILE* inputFile) {
2     Node* root = NULL;
3     char character;
4     char code[64];
5     uint32_t codeIndex = 0;
6     uint32_t readingCode = 0;
7     char lastChar = '\0';
8     uint32_t firstColonFound = 1;
9
10    while (1) {
11        uint32_t c = fgetc(inputFile);
12        if (c == EOF) break;
13
14        // Vérifier la séquence '---' pour arrêter la lecture
15        if (c == '-' && lastChar == '-' && codeIndex == 0) {
16            break;
17        }
18
19        // Si un saut de ligne est trouvé, on traite le code
20        if (c == '\n' && readingCode > 0) {
21            if (codeIndex > 0) {
22                code[codeIndex] = '\0';
23                construireHuffmanTree(&root, character, code);
24            }
25            codeIndex = 0;
26            readingCode = 0;
27            continue;
28        }
29
30        if (c == ':') {
31            long position = ftell(inputFile);
32            uint32_t tmp = fgetc(inputFile);
33            if (tmp == c) {
34                character = ':';
35            }
36            else {
37                fseek(inputFile, position, SEEK_SET);
38            }
39            readingCode = 1;
40            codeIndex = 0;
41            continue;
42        }
43
44        if (readingCode) {
45            code[codeIndex++] = c;
46        } else {
47            character = c;
```

```
48     }
49
50     lastChar = c;
51 }
52 return root;
53 }
```

Ce code montre comment les caractères et leurs codes binaires sont lus depuis le fichier, puis utilisés pour reconstruire l'arbre de Huffman. À chaque itération de la boucle, un caractère est extrait du fichier à l'aide de la fonction `fgetc(inputFile)`, qui retourne le prochain caractère du fichier ouvert. Dès qu'un caractère est trouvé, il est enregistré dans une variable (ici, `character`), et un processus de lecture du code binaire associé à ce caractère est déclenché.

Lorsque la lecture rencontre le caractère `' '`, cela marque le début de la lecture du code binaire associé au caractère précédent. Le code binaire est ensuite collecté caractère par caractère dans un tableau (ici `code[]`), jusqu'à ce qu'un saut de ligne (`' '`) soit rencontré, signalant la fin du code pour ce caractère. Ce code binaire est alors enregistré et associé au caractère correspondant. La fonction `construireHuffmanTree`, appelée après la collecte

du code, est responsable de l'insertion du nouveau nœud dans l'arbre de Huffman. Elle prend comme paramètres le caractère et le code binaire, et elle construit un nœud à partir de ces informations. Ce nœud est ensuite inséré dans l'arbre de manière appropriée en suivant la structure de l'arbre de Huffman. Plus précisément, le nœud est inséré de manière à ce que les caractères les plus fréquents soient plus proches de la racine de l'arbre, minimisant ainsi la longueur de leur code binaire. L'insertion dans l'arbre se

fait de manière récursive : si le code binaire commence par un `0`, on se déplace vers le sous-arbre gauche, et si le code commence par un `1`, on se déplace vers le sous-arbre droit. Chaque niveau de l'arbre correspond à un bit du code binaire, et chaque feuille représente un caractère de l'entrée originale. Le processus se répète pour chaque caractère

et son code binaire, jusqu'à ce que tout le fichier compressé soit parcouru et que l'arbre de Huffman complet soit reconstruit. Une fois l'arbre reconstruit, il peut être utilisé pour la décompression du fichier compressé, permettant de retrouver les caractères originaux en suivant les chemins dans l'arbre selon les codes binaires.

Remarque sur le choix de la taille du tableau `code[64]` :

- **Choix de la taille du tableau :** Le tableau `code[64]` est utilisé pour stocker les codes binaires associés à chaque caractère lors de la reconstruction de l'arbre de Huffman.
- **Principe de fonctionnement :**
 - Le code de Huffman est un code binaire variable, où chaque caractère est associé à un code binaire de longueur différente.
 - Le tableau `code` est utilisé pour lire et stocker ces codes binaires pendant la construction de l'arbre de Huffman.
- **Pourquoi 64 ?**
 - Le choix de 64 pour la taille du tableau est basé sur la longueur maximale des codes binaires générés par l'algorithme de Huffman.
 - En théorie, un code binaire peut atteindre une longueur maximale équivalente au nombre de caractères distincts dans le fichier, mais cela reste relativement rare.
 - La taille de 64 est largement suffisante pour couvrir la plupart des cas, où les codes binaires sont souvent bien plus courts.
- **Avantages :**
 - Une taille fixe de 64 simplifie la gestion de la mémoire et évite des opérations de réallocation dynamiques pendant l'exécution.
 - La taille de 64 est un compromis entre la consommation de mémoire et la garantie que tous les codes binaires seront stockés sans risque de débordement.

9.2 Construction de l'arbre de Huffman pour la décompression

L'arbre de Huffman pour la décompression est similaire à celui utilisé pour la compression, à la différence près qu'il ne contient pas de champ **frequency**. Voici les points clés :

- L'arbre de décompression est binaire, chaque nœud contient un caractère et son code binaire associé.
- Les nœuds internes servent de pointeurs vers les nœuds enfants, sans contenir de caractère.
- La racine de l'arbre est le point d'entrée pour la décompression.
- Chaque feuille contient un caractère, et la profondeur de chaque nœud détermine la longueur du code binaire.
- Contrairement à l'arbre de compression, le champ **frequency** n'est pas nécessaire dans l'arbre de décompression.
- La structure des nœuds de l'arbre de décompression est simplifiée, ne contenant que le caractère et un indicateur de son utilisation (champ `useData`).

Extrait de code de la construction de l'arbre de Huffman:

```

1 typedef struct node {
2     char data;           // Caractère dans le nœud
3     bool useData;        // Indicateur si le champ data est
                           // utilisé
4     struct node *left;   // Nœud gauche
5     struct node *right;  // Nœud droit
6 } Node;
7
8 void construireHuffmanTree(Node** root, char character, const
   char* code) {
9     if (*root == NULL) {
10         *root = createNode('\0', false); // Créer un nœud
           interne
11     }
12     Node* current = *root;
13     while (*code) { // Suivre le chemin du code binaire
14         if (*code == '0') {
15             if (!current->left) current->left = createNode('\0',
               false);
16             current = current->left;
17         } else if (*code == '1') {
18             if (!current->right) current->right = createNode('\0',
               false);
19             current = current->right;
20         }
21         code++;
22     }
23     current->data = character; // Stocke le caractère
24     current->useData = true;   // Marque le caractère comme
           utilisé
25 }

```

Ce code montre comment un caractère et son code binaire sont utilisés pour reconstruire l'arbre. Chaque code binaire détermine le chemin à suivre dans l'arbre, et à la fin, le caractère est stocké dans le nœud correspondant.

En résumé, l'arbre de Huffman pour la décompression est une version simplifiée de l'arbre de compression, qui ne prend pas en compte les fréquences des caractères, rendant la décompression plus rapide et directe.

9.2.1 Complexité de la construction de l'arbre

La construction de l'arbre de Huffman est relativement rapide, car elle nécessite de parcourir le fichier une seule fois pour lire les codes et les caractères associés. En moyenne, la complexité de cette étape est $O(n)$, où n est le nombre de caractères dans le fichier compressé. Cependant, l'insertion des nœuds dans l'arbre peut varier en fonction de la fréquence des caractères, ce qui influence la profondeur de l'arbre.

9.3 Décodage du fichier compressé

Une fois l'arbre reconstruit, la décompression du fichier compressé commence par la lecture bit à bit du fichier d'entrée. À chaque bit, l'algorithme fait un pas dans l'arbre de Huffman : si le bit est '0', l'algorithme se déplace vers le sous-arbre gauche, et s'il est '1', il se déplace vers le sous-arbre droit. Lorsqu'une feuille de l'arbre est atteinte (un nœud avec un caractère associé), ce caractère est ajouté au fichier décompressé. Le processus se répète jusqu'à ce que tout le fichier compressé ait été parcouru.

Voici le code qui effectue la décompression :

```

1 void decompressFile(const char* inputFilename, const char*
  outputFilename) {
2     FILE* inputFile = fopen(inputFilename, "rb");
3     FILE* outputFile = fopen(outputFilename, "wb");
4     if (!inputFile || !outputFile) {
5         perror("Erreur d'ouverture du fichier");
6         exit(EXIT_FAILURE);
7     }
8
9     // Construction de l'arbre de Huffman
10    Node* root = chargerHuffmanTree(inputFile);
11    Node* current = root;
12
13    uint8_t c;
14    while ((c = fgetc(inputFile)) != '\n' && c != EOF) {
15        // Ne rien faire, juste avancer caract re par caract re
16    }
17
18    // Lire les bits et parcourir l'arbre
19    uint32_t bit;
20    while ((bit = fgetc(inputFile)) != EOF) {
21        if (bit == '0') {
22            current = current->left;
23        } else if (bit == '1') {
24            current = current->right;
25        }
26
27        if (current->useData) { // Si on atteint une feuille
28            fputc(current->data, outputFile); // crire le
                caract re dans le fichier de sortie
29            current = root; // Revenir la racine de l'arbre
30        }
31    }
32
33    fclose(inputFile);
34    fclose(outputFile);
35    freeHuffmanTree(root);
36    printf("Fichier d compress avec succ s !\n");
37 }

```

Cette fonction lit le fichier compressé, décode les bits en parcourant l'arbre de Huffman et écrit les caractères décompressés dans le fichier de sortie.

9.4 Libération de l'arbre de Huffman

Il est important de libérer l'arbre de Huffman une fois que la décompression est terminée pour éviter les fuites de mémoire. La fonction suivante permet de libérer l'arbre après l'utilisation :

```
1 void freeHuffmanTree(Node* root) {  
2     if (root) {  
3         freeHuffmanTree(root->left);  
4         freeHuffmanTree(root->right);  
5         free(root);  
6     }  
7 }
```

Cela permet de nettoyer la mémoire allouée dynamiquement pour les nœuds de l'arbre.

9.5 Résumé sur les détails du programme

9.5.1 Structures de données

- **Node** : Représente un nœud de l'arbre de Huffman. Chaque nœud contient un caractère et un indicateur pour déterminer si le champ caractère est utilisé, ainsi que des pointeurs vers ses fils gauche et droit.
- **tab** : Structure qui contient un tableau de nœuds représentant les éléments à insérer dans l'arbre de Huffman.

9.5.2 Fonctions principales

- **createNode** : Crée un nœud avec un caractère donné, un indicateur pour marquer si le caractère est utilisé, et des pointeurs vers les fils gauche et droit.
- **construireHuffmanTree** : Construit l'arbre de Huffman en insérant les caractères et leurs codes binaires dans l'arbre à l'aide des chaînes de bits.
- **chargerHuffmanTree** : Charge l'arbre de Huffman à partir d'un fichier en lisant les codes associés aux caractères.
- **freeHuffmanTree** : Libère la mémoire allouée pour l'arbre de Huffman après son utilisation.
- **decompressFile** : Fonction principale pour décompresser un fichier en lisant les bits compressés et en les utilisant pour parcourir l'arbre de Huffman, en reconstruisant le fichier original.

10 Instructions pour l'utilisation

10.1 Compiler le programme

Utilisez un compilateur C pour compiler le programme. Par exemple, avec GCC, vous pouvez exécuter la commande suivante :

```
gcc -o huffman_decompression huffman_decompression.c
```

10.2 Exécuter le programme

Le programme prend en argument le nom du fichier à décompresser et le nom du fichier de sortie pour stocker le fichier décompressé. Vous devez fournir ces deux arguments lors de l'exécution.

La syntaxe de la commande est la suivante :

```
./huffman_decompression <nom_du_fichier_input> <nom_du_fichier_output>
```

Exemple :

```
./huffman_decompression fichier_compressé.bin fichier_decompressé.txt
```

10.3 Résultat attendu

Une fois l'exécution terminée, le programme décompresse le fichier d'entrée et sauvegarde le fichier décompressé dans le fichier de sortie spécifié. Un message de confirmation s'affiche :

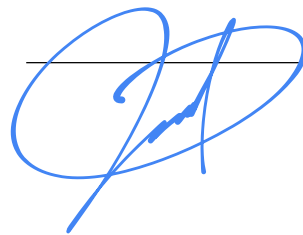
```
Fichier décompressé avec succès dans : fichier_decompressé.txt
```

Engagement de non-plagiat

Nous, soussigné **SEBA**, étudiants en **1er Année Ingénieur en Informatique** à l'école d'ingénieur à **Sup Galilée**, déclarons être pleinement conscient(e)s que la copie de tout ou partie d'un document, quel qu'il soit, publié sur tout support existant, y compris sur Internet, constitue une violation du droit d'auteur ainsi qu'une fraude caractérisée, tout comme l'utilisation d'outils d'Intelligence Artificielle pour générer une partie de ce rapport ou du code associé.

En conséquence, nous déclarons que ce travail ne comporte aucun plagiat, et assurons avoir cité explicitement, à chaque fois que nous en avons fait usage, toutes les sources utilisées pour le rédiger.

Fait à Paris, le 01/02/25.



Signature