Compression de Huffman

Programme de ce TP

- Les fichiers
- Les listes chaînées
- Les arbres binaires
- La compression

Consignes

Dans ce TP, vous ne devez rendre **uniquement** que les fichiers sources (`.cs`) de vos programmes.
La structure du rendu doit être comme ceci :

```
/rendu-tp-login_x.zip
   |-- login_x /
          |-- AUTHORS
          |-- Compression.cs
          |-- Decompression.cs
          |-- Misc.cs
          |-- README
```

`rendu-tp-login_x.zip`	nom de l'archive à rendre
`AUTHORS`	vous devez remplacer le contenu du fichier par une étoile `*`, un espace, puis votre login (ex : `login_x`) suivi d'un retour à la ligne
`README`	vous devez écrire dans ce fichier, tout commentaire sur le TP, votre travail ou bien plus généralement sur vos points forts/faibles. un fichier `README` vide sera considéré comme une archive sale (malus).

Il va de soi que toutes les occurrences de `login_x` sont à remplacer par **votre** login.

Tout le TP est à réaliser en mode console : `Fichier` ⇒ `Nouveau` ⇒ `Projet` ⇒ `Application Console`

TESTEZ VOTRE CODE !
Vous devez rendre un code qui compile, sans erreur ni warning !
Autrement votre rendu ne sera pas corrigé !

1 Algorithme de Huffman

1.1 Description générale de l'algorithme

Au cours de ce TP, vous allez pouvoir réaliser votre logiciel de compression de fichiers textes. Il existe beaucoup d'algorithmes de compression de données plus ou moins efficaces. Dans le cadre de ce TP nous allons nous intéresser à l'algorithme d'Huffman qui offre un taux de compression qui varie entre `40%` et `60%`.

A l'état normal, tous les caractères d'un fichier texte sont codés sur 8 bits et ce peu importe la fréquence d'apparition des caractères. Ainsi un fichier contenant le texte `ABCDEF` aura une taille de `6 × 8 = 48` bits tout comme le fichier contenant `AAAAAA`.

L'algorithme d'Huffman va chercher à modifier le codage des caractères pour coder les caractères qui apparaissent le plus fréquemment sur 2 ou 3 bits, tandis que les caractères qui apparaissent très rarement seront codés sur plus de bits. En reprenant notre exemple précédent, le fichier contenant `AAAAAA` aura alors une taille de `2 × 6 = 12` bits.

Pour arriver à un tel résultat, l'algorithme va utiliser un arbre binaire où chaque feuille de l'arbre va contenir en élément un caractère. Le codage d'un caractère va alors être déterminé par un parcours de l'arbre jusqu'à la feuille contenant le caractère recherché. A chaque fois que l'on passe par un fils gauche, on ajoute le bit `0` au codage du caractère, et lorsqu'on passe par un fils droit on ajoute le bit `1`. Voici un exemple pour mieux comprendre le fonctionnement de l'arbre :

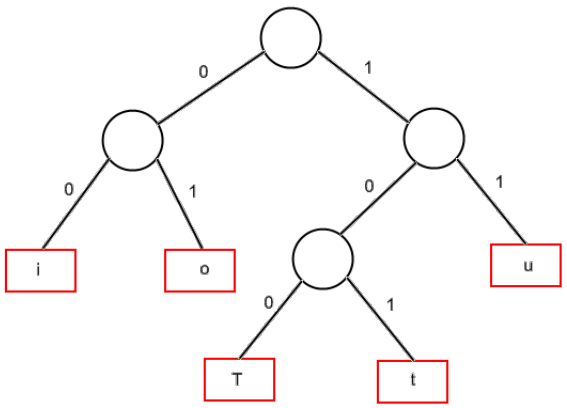

FIGURE 1 – Arbre binaire de Huffman

Dans cet exemple, la lettre `i` sera codé sur 2 bits `00`, la lettre `T` sera codé sur 3 bits `100` et ainsi de suite. On pourra donc coder le mot `Titou` en utilisant ce codage et obtenir un fichier d'une taille de `12` bits au lieu des `40` bits nécessaires en temps normal.

La difficulté est donc dans la construction d'un tel arbre. En effet, il n'est pas organisé n'importe comment. Premièrement, tous les caractères doivent être sur une feuille afin d'éviter les ambiguïtés dans le codage. Et pour que l'algorithme soit réellement efficace, il faut deuxièmement faire en sorte que les caractères qui sont utilisés le plus souvent se situe en haut de l'arbre afin d'avoir un codage plus court. De plus, pour décompresser un fichier compressé avec cette technique, il est nécessaire de sauvegarder l'arbre pour pouvoir décoder les caractères.

1.2 Déroulement de l'algorithme

Voici un descriptif des différentes étapes par lequel passe l'algorithme pour compresser et décompresser un fichier.

Étapes lors de la compression :
- construction de la table des fréquences des caractères
- construction de l'arbre binaire
- construction de la table de correspondance caractère - code
- compression du fichier texte
- sauvegarde de l'arbre binaire

Étapes lors de la décompression :
- reconstruction de l'arbre
- décompression du fichier

1.3 Objectifs

Vous allez devoir coder les étapes de la compression et de la décompression.

On vous fournit le fichier `Misc.cs` contenant les types à utiliser, il est interdit de le modifier.
On vous fournit également les deux fichiers `Compression.cs` et `Decompression.cs` à remplir.
On ne s'occupera que des caractères de la table ASCII simple (0-255).

2 Compression

Fichier à rendre : `Compression.cs`

2.1 Table des fréquences

Pour s'échauffer, quelque chose de simple : la construction de la table des fréquences des caractères d'un fichier texte. Pour cela, vous allez utiliser le type `Misc.TFrequency` fournis dans le fichier `Misc.cs`. Le type `Misc.TFrequency` est un tableau de 255 cases contenant des `int`. L'index `i` vas symboliser le code ASCII des caractères et l'élément contenu dans la case va correspondre au nombre de fois où le caractère est apparu dans le fichier texte.

Codez la fonction `CreateFrequency` dans le fichier qui prend en argument le nom d'un fichier et retourne un `Misc.TFrequency` correctement remplie.

2.2 Création de l'arbre binaire

Un peu plus compliqué : la création de l'arbre binaire en respectant les contraintes de l'algorithme. La construction de l'arbre binaire va passer par la création d'une liste chainée. Chaque élément de cette liste contiendra un arbre associés à son poids correspondant à la somme des fréquences des caractères contenus dans cet arbre. Elle sera triée par ordre croissant en fonction du poids. Pour construire l'arbre, il ne restera plus qu'à associer les deux sous arbres de poids minimum ensemble et d'insérer dans la liste le nouvel arbre créé et de réitérer jusqu'à ce qu'il ne reste plus qu'un élément dans la liste.

Vous devrez utiliser les types `Misc.TList` et `Misc.TBinaryTree` pour créer votre liste chainée et arbre.

2.2.1 Passage par les listes chainées

Dans un premier temps, vous allez coder la fonction `CreateList` qui prend en argument un `Misc.TFrequency` et qui retourne un `Misc.TList` triée par ordre croissant par rapport aux poids. Initialement, cette liste contiendra uniquement les feuilles de l'arbre binaire c'est à dire que tous les attributs `Left` et `Right` seront à `null`. Pensez qu'il n'est pas utile d'insérer dans la liste des caractère qui ont une fréquence nulle !

Voici un exemple d'une liste chainée construite à partir d'une table de fréquence :

Z	E	K	D	O	L	W	R
3	20	5	14	17	12	1	12

FIGURE 2 – Table de fréquence

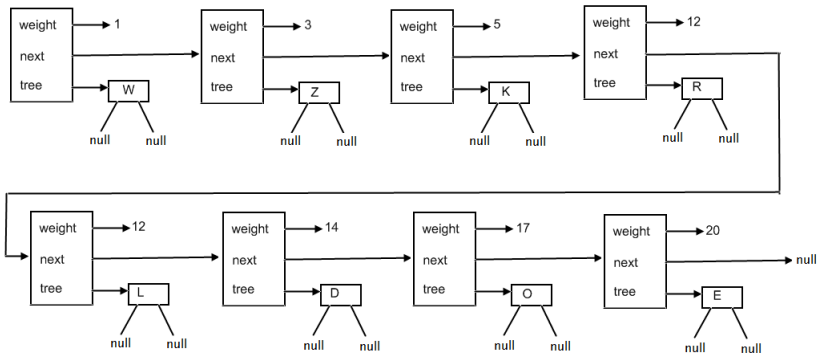

FIGURE 3 – Liste chainée construite

2.2.2 Construction de l'arbre

Une fois que la liste chainée est initialisée, vous allez devoir créer l'arbre binaire. Voici la méthode que vous allez devoir suivre et qui est illustrée par les figures `4` à `10` :

- choisir les deux éléments ayant la plus petite valeur à `weight`
- créer un nœud de l'arbre avec en fils gauche, l'arbre associé à la plus petite valeur de `weight` et en fils droit l'arbre associé à la deuxième plus petite valeur de `weight`
- insérer au bon endroit dans la liste le nouveau nœud d'arbre avec pour valeur à `weight` la somme des `weight` des deux éléments traités et supprimer de la liste les deux éléments traités
- réitérer cela jusqu'à ce qu'il n'y ait plus qu'un seul élement dans la liste : l'arbre binaire sera l'arbre associé a l'attribut `tree` de l'élément restant

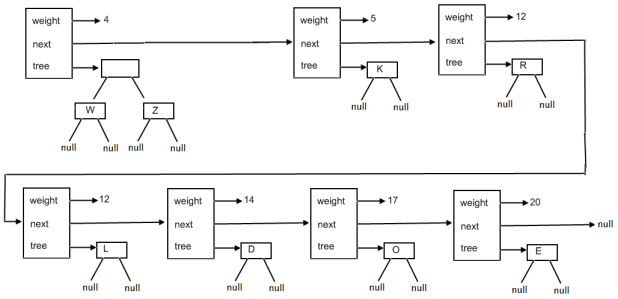

FIGURE 4 – Itération 1

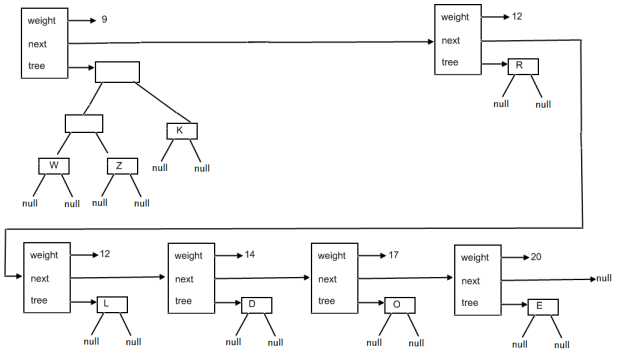

FIGURE 5 – Itération 2

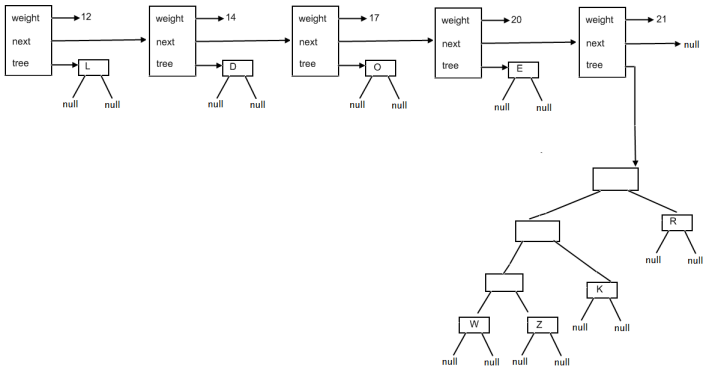

FIGURE 6 – Itération 3

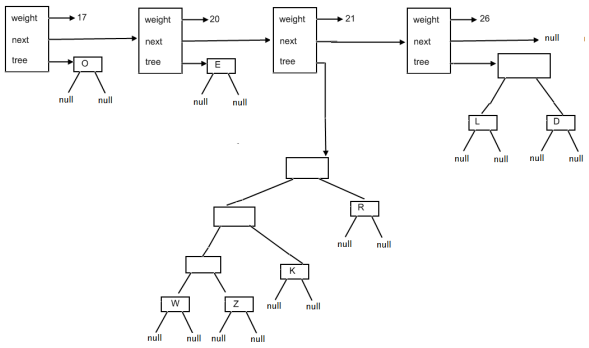

FIGURE 7 – Itération 4

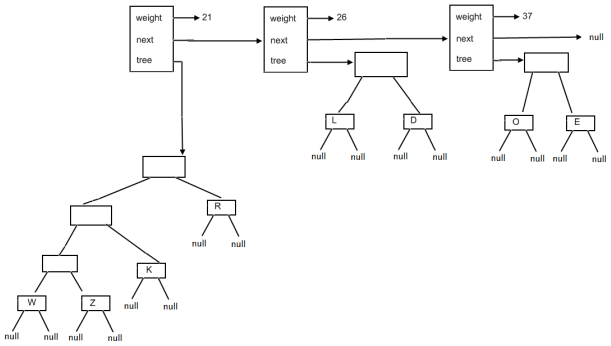

FIGURE 8 – Itération 5

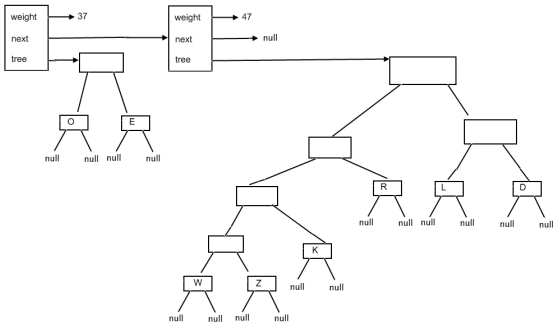

FIGURE 9 – Itération 6

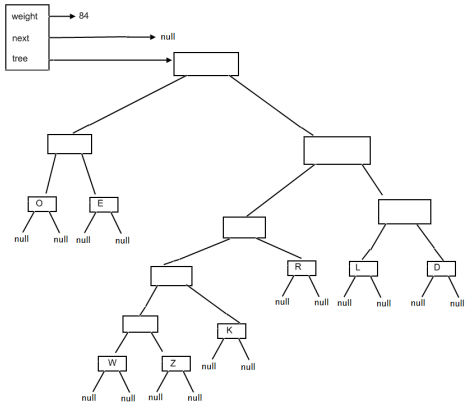

FIGURE 10 – Itération 7

2.3 Table de correspondance

Pour construire la table de correspondance, vous allez utiliser le type `Misc.TMatch`. Il s'agit d'un tableau de `string` encore une fois de 255 cases où l'index va symboliser le code ASCII des caractères. La `string` qui sera contenue dans chaque case correspondra à une succession de `0` et de `1` selon le parcours de l'arbre nécessaire pour atteindre le caractère, `0` correspondant à un parcours sur le fils gauche et `1` correspondant à un parcours sur le fils droit.

Pour construire facilement cette table de correspondance, vous allez coder un parcours en profondeur récursif d'un arbre binaire. Vous devez coder une fonction `CreateMatch` qui prend en argument un arbre `Misc.TBinaryTree` et qui retourne une table `Misc.TMatch` et qui appellera une procédure `CreateMatchRec` qui prend en argument une table `Misc.TMatch`, un arbre `Misc.TBinaryTree` et une `string` et qui effectue un parcours profondeur de l'arbre en mettant à jour la string et la table.

Voici la table de correspondance que l'on doit obtenir à partir de l'arbre binaire en exemple de l'exercice 2 (sans tenir compte des caractères n'étant pas sur l'arbre).

Z	E	K	D	O	L	W	R
10001	01	1001	111	00	110	10000	101

FIGURE 11 – Table de correspondances

2.4 Création d'un fichier intermédiaire

Afin de vous simplifier la tâche, vous n'allez pas coder directement la compression complète mais juste créer un fichier intermédiaire qui sera compresser par une fonction ultérieure.

Ce fichier sera créé de la façon suivante : vous allez lire le fichier initial, et vous allez remplacer dans le nouveau fichier chaque caractère par la chaine de caractères correspondante dans la table de correspondance créée lors de l'exercice précédent. Vous appellerez ce fichier avec le même nom que le fichier initial mais portant l'extension `.tmp`.

Vous allez coder la procédure `CreateTemp` qui prend en argument le nom du fichier à compresser et une table de correspondance `Misc.TMatch` et qui créé le fichier `.tmp`.

2.5 Compression du fichier

Pour finir la compression, vous allez coder une procédure `Compress` qui prend en argument le nom d'un fichier et qui le compresse au format `.mzip`. Vous allez devoir sauvegarder la table de correspondance (attention au caractère `\n` - retour à la ligne). Pour chaque caractère de notre table nous allons stocker le caractère, suivis de sa correspondance, puis le caractère `*`. Nous allons stocker les caractères 1-9 puis 11-255 et enfin le 10 (`\n`, noté `$$`). Un fois la correspondance écrite, nous allons mettre un retour à la ligne avant de stocker notre fichier source convertit sous sa forme binaire compressée.

3 Décompression

Fichier à rendre : `Decompression.cs`

3.1 Recréer le fichier texte

Vous allez commencer la décompression des fichiers à partir des fichiers `.tmp` que vous générez dans la compression et des arbres binaires que vous avez créé. Vous allez donc coder la procédure `CreateText` qui prend en argument le nom d'un fichier à décompresser SANS son extension `.tmp` ainsi qu'un arbre binaire `Misc.TBinaryTree` et qui génère un fichier `.txt` portant le même nom que le fichier à décompresser. On considère que le nom de fichier passé en argument est correct et qu'il existe bien un fichier portant l'extension `.tmp` avec ce nom.

Pour générer le fichier texte vous allez devoir parcourir l'arbre binaire par rapport aux caractères `0` ou `1` que vous allez rencontrer dans le fichier `.tmp`. L'algorithme est le suivant :

- Initialement vous êtes au début du fichier et à la racine de l'arbre
- Tant que vous n'êtes pas à la fin du fichier vous lisez des `0` ou des `1` dans le fichier
- Si vous lisez un `0` vous descendez sur le fils gauche, sinon sur le fils droit
- Si vous arrivez sur une feuille (fils gauche et fils droit sont à `null`), vous écrivez dans le fichier `.txt` le caractère correspondant et vous retournez sur la racine de l'arbre
- Vous itérez cela tant que vous n'avez pas lu tout le fichier

3.2 Procédure de décompression

Afin de pouvoir utiliser votre procédure à partir des fichiers compressés `.mzip`, vous devez coder une procédure `Uncompress` prend en argument un nom de fichier à décompresser SANS extension, qui va générer le fichier `.tmp` et retourner l'arbre binaire associé. Passez par une fonction qui se charge de reconstruire l'arbre, et une qui convertit le binaire compressé en binaire lisible utilisable par la fonction de l'exercice précédent.

4 Bonus

La ligne de commande c'est bien, mais le mode graphique c'est mieux :)