# Projet — Codage de Huffman

## Les consignes

Le projet est à réaliser en **binôme**. Le rendu devra se faire avant le **Jeudi 11 Octobre 23h59** sur Celene exclusivement. Le rendu devra comprendre **exactement** deux fichiers :

- le code source (et tous les fichiers nécessaires à l'éxécution) correctement archivé au format `tar.gz`.
- un rapport (deux-trois pages maximum) au format Pdf, expliquant la structure de votre code **et** son utilisation.

L'archive ne devra comprendre que les fichiers source, et contenir tous les éléments nécessaires à la compilation (`Makefile` notamment) **et** à l'exécution.

## Introduction *Le contexte*

Pour représenter de l'information, un codage est utilisé. L'un des codages les plus standards pour représenter des caractères est l'American Standard Code for Information Interchange (couramment appelé ASCII). Il définit la façon dont $2^7 = 128$ symboles sont représentés. Ainsi, la lettre `A` se code `1000001` (soit 65 en décimal), la lettre `Z` est codée par `1011010`, ...

Il est également possible de définir un codage personnalisé. Si l'on considère une séquence d'ADN, seuls les symboles `A`, `C`, `G`, et `T` sont pertinents car ils correspondent aux quatres nucléotides (Adenine, Cytosine, Guanine, Thymine). Soit le codage suivant :

```
A  00
C  01
G  10
T  11
```

Le nombre de bits nécessaires pour représenter une séquence d'ADN de longueur $l$ est donc $2 * l$. Soit maintenant le codage suivant :

```
A  000
C   1
G   01
T  001
```

Pour déterminer le nombre total de bits nécessaires pour représenter une séquence d'ADN, il faut désormais connaître le nombre d'occurrences $\#$`c` de chaque caractère `c`. Ici :

$$3 * \#A + \#C + 2 * \#G + 3 * \#T$$

C'est sur ce principe que repose le codage de **Huffman**.

## Codage de Huffman *Coder et décoder*

Les codages de Huffman sont utilisés pour **compresser des données**. C'est une compression sans perte d'information (contrairement à JPEG ou MP3 par exemple, où l'information est dégradée, ce qui occasionne une perte de qualité). Pour compresser, un codage à longueur variable est utilisé pour représenter chacun des symboles de la source à compresser. Le codage est déterminé selon la fréquence des symboles de la source : un code court pour un symbole fréquent, un code plus long pour ceux moins fréquents.

Dans la suite, seuls des **codes préfixes** seront considérés. Ces codes sont un peu particulier puisqu'aucun mot de code n'est aussi **préfixe** d'un autre mot de code. Cela n'affaiblit pas la compression et facilite grandement la

décompression. Ainsi, le mot binaire `0111001011` se décode de façon **unique** par `GCCTGC` avec code préfixe suivant :

```
A  000
C   1
G   01
T   001
```

Si le caractère `A` avait été codé par `00`, le code ne serait plus préfixe puisque `00` est préfixe de `001` qui code `T`. Ainsi, la décompression aurait pu amener au texte `GCCACGC` **ou** `GCCTGC`.

## Représentation d'un codage

*Fake Plastic Trees*

Pour représenter un code préfixe, il est nécessaire d'utiliser les **arbres binaires**. Ces structures de données sont composées de **noeuds** possédant chacun une valeur, un fils gauche (éventuellement `NULL`) et un fils droit (éventuellement `NULL`). Les noeuds n'ayant aucun fils sont appelés des **feuilles**. Un arbre binaire est identifié par un noeud particulier, appelé **racine**.

De nombreuses primitives peuvent être associées à une telle structure de données, parmi lesquelles :

(i) création et destruction d'un arbre
(ii) ajout et suppression d'un noeud
(iii) recherche d'un élément dans un arbre
(iv) ...

Cette liste n'est évidemment pas exhaustive, et peut également contenir des fonctions inutiles au bon fonctionnement du projet. Un exemple d'arbre binaire est donné Figure 1.

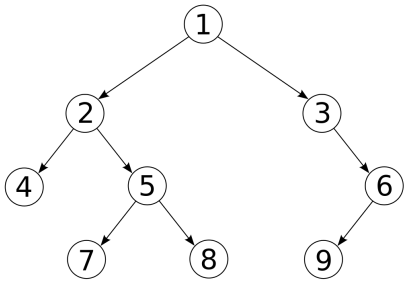



FIGURE 1 – Un arbre binaire de racine 1 et de feuilles $\{4, 7, 8, 9\}$. ©Wikipedia

$\mathcal{Q}_1$. Implémenter les fichiers nécessaires permettant de gérer des arbres binaires.

**Un peu de généricité ?**

S'il est possible de réaliser ce projet en n'utilisant une structure d'arbres binaires n'acceptant qu'un seul type de noeud, une attention particulière sera portée au code permettant de créer des arbres binaires **génériques**.

## Codage de Huffman

*Calculer le code préfixe*

En utilisant les structures précédemment définies, l'objectif est maintenant de construire un arbre représentant un code préfixe. Étant donné un texte, la première étape consiste à calculer la fréquence de chacun des symboles composant ce texte. Ensuite, **l'algorithme de Huffman** réalisera la construction d'un arbre préfixe optimal.

Les arbres utilisés dans le codage de Huffman sont en fait **complets** : chaque noeud de l'arbre a soit 0 fils (et on dit que c'est une *feuille*), soit 2 fils (appelés *fils gauche* et *fils droit*).

**Définition**

Un arbre binaire complet est défini comme :

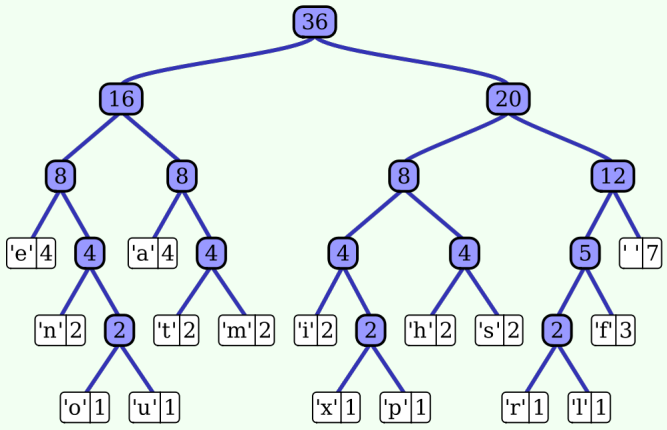
- une feuille ; ou
- un noeud ayant comme fils gauche un arbre $A_g$ et comme fils droit un arbre $A_d$.

**Description de l'algorithme.** Soient $T$ un texte contenant $n$ symboles et $f : \{1, \ldots, n\} \to \mathbb{N}$ une *fonction de fréquence*, associant à chaque symbole $c \in \{1, \ldots, n\}$ de $T$ son nombre d'occurences $f(c)$. L'algorithme commence par créer $n$ arbres minimalistes : initialement, chacun de ces arbres ne contient qu'un seul noeud. Ce noeud est à la fois la **racine** et **l'unique feuille** de l'arbre. Il porte deux informations : le symbole et sa fréquence.

Dans la suite, la **fréquence** d'un arbre désignera la fréquence contenue à sa **racine**. Pour les arbres minimalistes, leur fréquence est la fréquence contenue dans leur unique noeud. L'algorithme maintient une *liste* d'arbres, initialisée au début à partir des $n$ arbres minimalistes correspondant aux symboles du texte.

Tant que la liste d'arbres contient au moins deux arbres, deux des arbres de plus petite fréquence en sont extraits pour être **fusionnés**. Une illustration complète de l'algorithme est donnée ci-dessous.

This is an example of a huffman tree — ©Wikipedia




**Choix des arbres à fusionner — exemple non contractuel**

S'il existe plusieurs possibilités pour choisir les deux arbres à fusionner, prendre ceux dont les symboles sont les plus petits dans l'ordre lexicographique — alphabétique.

Soient $A_1$ et $A_2$ ces nouveaux arbres. À chaque étape, il faut donc créer un nouvel arbre, composé d'un noeud dont les deux fils sont $A_1$ et $A_2$ et dont la fréquence est égale à la somme des fréquences de la racine de $A_1$ et de la racine de $A_2$. Ce nouvel arbre ainsi obtenu est ajouté à la liste. La longueur de la liste d'arbres diminuant à chaque étape (deux arbres sont extraits et un arbre est ajouté), l'algorithme termine toujours.

$\mathcal{Q}_2$. Implémenter les fichiers permettant de calculer un arbre binaire complet représentant un code préfixe pour une chaîne de caractères donnée.

**Création du code préfixe — compression.** Une fois l'arbre obtenu, il est nécessaire de le parcourir et d'affecter une valeur binaire (0 ou 1) à chaque branche afin d'en déduire un code. La règle à utiliser est la suivante :

en démarrant de la racine, un déplacement à gauche est modélisé par un 0, un déplacement à droite par un 1.

Sur l'exemple, le code du caractère `t` est ainsi `1001`, et celui du caractère `x` est `01101`.

$\mathcal{Q}_3$. Implémenter une méthode qui, étant donnée un texte retourne la séquence compressée associée.

**Décompression.** L'utilisation des codages préfixes permet de décoder de manière **simple** et **unique** un texte compressé avec l'algorithme de Huffman. Il s'agit simplement de se déplacer dans l'arbre représentant le code préfixe afin de savoir quels symboles sont codés par la séquence compressée, et ainsi retrouver le texte original.

$\mathcal{Q}_4$. Implémenter une méthode qui, étant donnée une séquence compressée retourne le texte original associé.

## Gestion de fichiers

*Vingt mille lieues sous les mers*

Pour pouvoir appliquer correctement l'algorithme de Huffman, il est nécessaire d'adapter le code afin de pouvoir compresser des fichiers texte en utilisant l'algorithme de Huffman précédemment codé. Un fichier est mis à disposition sur Celene, mais le choix est laissé libre.

$\mathcal{Q}_5$. Modifier les méthodes de compression et de décompression pour qu'elles puissent lire et écrire les données dans un fichier. La fonction devra également fonctionner depuis les sorties standards `stdout` et `stdin`.

## Algorithme générique

*Huffman template algorithm*

L'algorithme de Huffman peut se généraliser en autorisant à avoir d'autres fonctions de **poids** associées au code. La version présentée ci-dessus utilise les fréquences de chaque caractère comme poids. Il est en fait possible d'utiliser **n'importe quelle** fonction de poids, du moment qu'il est possible de les ordonner et de les regrouper entre eux (l'*addition* a été utilisée).

### Algorithme de Huffman générique

$\mathcal{Q}_6$. Implémenter l'algorithme de Huffman générique. La fonction de poids et l'opération (binaire) appliquée sur ces derniers doivent être génériques.

### Le rendu

Pour simplifier les tests du code rendu, l'algorithme de Huffman générique sera implémenté en plus de l'algorithme de Huffman classique.