Université Claude Bernard – Lyon1 — Licence Sciences et Technologies – L3
Algorithmique, Programmation et Complexité – LIFAP6 — Printemps 2017

# Examen de TP : codage de Huffman

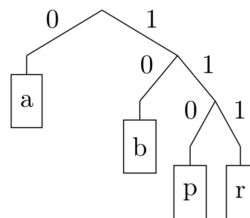
Durée : 3 heures.

## 1 Codage de Huffman

Le codage de Huffman est un procédé très utilisé en compression de données. Il sert à encoder un texte en binaire, en utilisant pour chaque lettre un nombre de bits dépendant du nombre de fois où la lettre est présente : plus la lettre apparaît, plus le nombre de bits est petit. Ainsi, le nombre total de bits utilisés pour encoder le texte est réduit par rapport à un codage ASCII standard qui utilise huit bits pour chaque lettre.

### 1.1 Arbres binaires

Le code binaire associé à chaque lettre est défini par un arbre binaire. Les feuilles de l'arbre correspondent aux lettres de l'alphabet. Les nœuds interne ne contiennent pas d'information. Le chemin emprunté pour atteindre une feuille depuis la racine définit le code de la lettre sur cette feuille : à gauche 0, à droite 1. Par exemple :




a → 0
b → 10
p → 110
r → 111

barbapapa → 10011110011001100

### 1.2 Optimisation du code

Lorsque le texte à encoder est connu, il est possible d'optimiser l'arbre binaire utilisé pour raccourcir les codes des lettres les plus fréquentes. C'est ce que fait l'algorithme de Huffman : il commence par compter dans le texte le nombre d'apparitions de chaque caractère. Par exemple, dans la phrase :

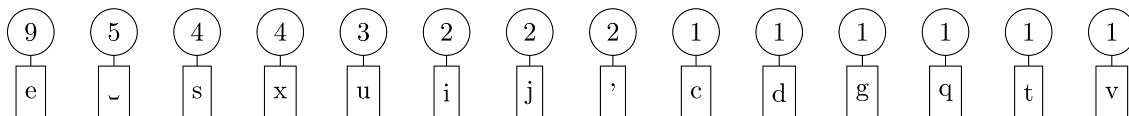
je veux et j'exige d'exquises excuses

nous avons :

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| e : | 9 | s : | 4 | u : | 3 | j : | 2 | c : | 1 | g : | 1 | t : | 1 |
| ␣ : | 5 | x : | 4 | i : | 2 | ' : | 2 | d : | 1 | q : | 1 | v : | 1 |

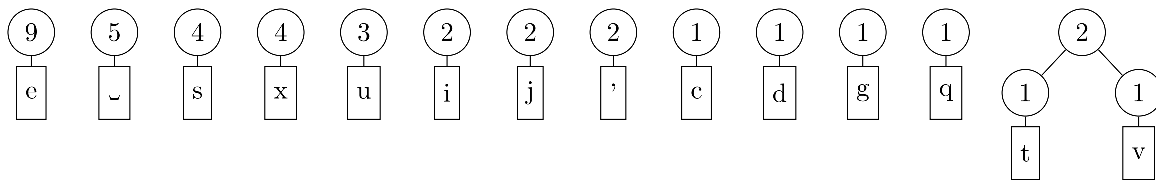
où « ␣ » est le caractère espace, et « ' » est l'apostrophe.

À partir de ces fréquences, l'algorithme initialise un arbre par caractère, avec comme poids le nombre d'apparitions du caractère :

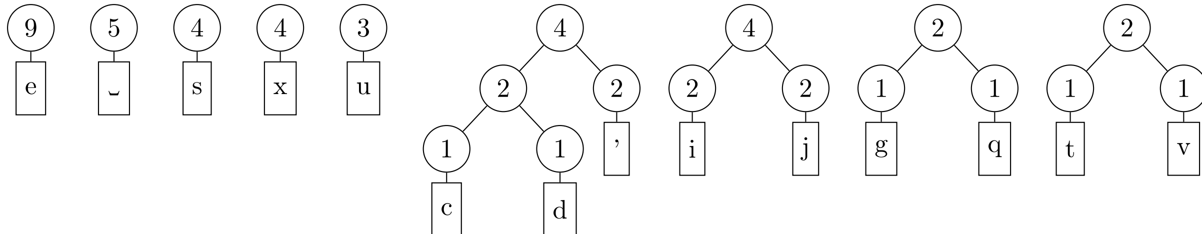



Dans la suite, nous indiquerons le poids d'un sous-arbre au niveau du nœud interne où il est enraciné.

À chaque itération, l'algorithme sélectionne les deux arbres ayant les poids les plus faibles, et les assemble pour former les deux enfants d'un arbre binaire dont le poids est la somme des deux arbres. Lorsque plusieurs arbres sont à égalité pour le poids le plus faible, l'arbre sélectionné parmi eux peut être n'importe lequel. Ainsi, à la première itération, nous pourrions obtenir :




et après quatre itérations de plus :




Lorsqu'il ne reste plus qu'un seul arbre, l'algorithme est terminé. Plus formellement, l'algorithme de Huffman est le suivant :

**Fonction** Huffman → arbre
- **données :** l'alphabet et les fréquences de chaque lettre
- **résultat :** l'arbre binaire du codage de Huffman
- **Algorithme**
  - créer une file à priorités
  - **pour chaque** caractère $c$ de l'alphabet **faire**
    - créer un arbre `a`
    - `a`.caractère ← $c$
    - `a`.poids ← fréquence de $c$
    - ajouter `a` à la file à priorités
  - **tant que** la file à priorités contient plus d'un arbre **faire**
    - créer un arbre `a`
    - `a`.gauche ← l'arbre de poids le plus faible de la file à priorités (que l'on retire de la file)
    - `a`.droite ← l'arbre de poids le plus faible de la file à priorités (que l'on retire de la file)
    - `a`.poids ← `a`.gauche.poids + `a`.droite.poids
    - ajouter `a` à la file à priorités
  - **retourner** le seul arbre contenu dans la file à priorités

## 2 Base de code

Nous vous fournissons une archive de base, munie de tous les fichiers à remplir, du Makefile pour les compiler. Vous n'avez pas normalement besoin d'ajouter de fichiers supplémentaires.

## 3 Modalités de rendu

Votre code est a rendre sous la forme d'une archive via Tomuss `https://tomuss.univ-lyon1.fr`. Le Makefile dispose d'une règle pour générer l'archive pour vous :

```
make archive
```

Cette commande crée une archive avec votre nom d'utilisateur comme nom, et y intègre le Makefile ainsi que tous les fichiers `cpp` et `hpp`.

## 4 Liberté d'action et conseils

Pour ce TP, vous êtes autorisés à utiliser toutes les fonctionnalités de la librairie standard. Vous êtes également autorisés à consulter la documentation en ligne. Vous n'êtes par contre pas autorisés à communiquer entre vous, oralement ou via votre poste de travail ou votre téléphone. Le code que vous produirez devra rester le votre. Vous ne recevrez pas de points pour une implémentation tout prête récupérée en ligne. Cela ne vous empêche pas de vous en inspirer, mais prenez garde à mentionner vos sources car ne pas le faire constitue un plagiat.

Nous sommes conscients que quel que soit le soin que nous mettons à la rédaction du sujet, il est probable que certaines questions ne soient pas claires pour vous. N'hésitez donc pas à demander à votre encadrant de clarifier une question pour vous. De même il est parfois difficile d'évaluer la difficulté et le temps nécessaire pour répondre aux questions. Selon les rendus que nous recevrons, nous en tiendrons compte dans la notation en notant éventuellement sur plus de 20.

Ne jetez pas votre code s'il ne marche pas. Laissez le en commentaire et mentionnez dans vos commentaires ce que vous avez fait pour essayer de trouver l'erreur, éventuellement les messages d'erreur qui vous semblent significatifs et ce que vous en avez déduit. Le bonnes choses présentes dans ces commentaires pourront ainsi éventuellement vous bénéficier.

# 5 Votre travail

## 5.1 Structure de données de base

Pour représenter un arbre de Huffman, vous complèterez la structure présente dans le fichier `huffman.hpp`. La structure `HArbre` représente à la fois un nœud de l'arbre et l'arbre tout entier, donné par sa racine. Chaque nœud comporte :

— l'adresse de son enfant gauche ;
— l'adresse de son enfant droit ;
— un poids ;
— un caractère (qui n'est utilisé que si le nœud est une feuille).

## 5.2 Initialisation d'un nœud

Le squelette de code fourni comporte trois fonctions d'initialisation des nœuds. Ces fonctions réalisent l'allocation du nouveau nœud et renvoient son adresse.

### 5.2.1 Initialisation à partir d'un caractère

Cette fonction est utilisée pour initialiser les feuilles de l'arbre. Le poids à fournir est le nombre de fois où le caractère est apparu dans le texte à partir duquel l'arbre est créé. La fonction réalise l'allocation du nœud sur le tas, initialise ses champs et renvoie l'adresse du nœud créé.

### 5.2.2 Initialisation à partir des enfants

Cette fonction sert à construire l'arbre selon le principe de Huffman. À partir des deux sous-arbres à assembler, un nouveau nœud est alloué pour leur servir de parent, et pour encoder le poids de l'arbre résultant. Ce nœud ne représente pas de caractère particulier, seules les feuilles correspondent à un caractère. Comme pour l'initialisation à partir d'un caractère, le nouveau nœud est alloué sur le tas et son adresse est renvoyée par la fonction.

### 5.2.3 Initialisation à partir d'une chaîne de caractères

Cette fonction est celle qui construit réellement l'arbre via l'algorithme de Huffman. Elle prend en paramètre la chaîne de caractères à encoder, calcule le nombre d'apparitions de chaque lettre, initialise les feuilles de l'arbre, puis réalise la fusion progressive des arbres jusqu'à obtenir un arbre unique. C'est l'adresse du nœud racine de cet arbre qui est renvoyée. Nous vous conseillons de procéder par étapes :

— lister l'ensemble des caractères présents ainsi que leur nombre d'apparitions ;
— initialiser les feuilles de l'arbre et les insérer dans la file a priorité ;
— réaliser la fusion progressive des arbres.

Pour chacune de ces étapes, vérifiez bien que le résultat est correct avant de passer à la suivante.

La chaîne de caractères en entrée est fournie via le type `std::string`. Ce type se comporte comme un tableau, avec quelques outils en plus. Vous pouvez récupérer la taille de la chaîne de caractères via

```
int taille = s.size() ;
```

et vous pouvez accéder au i$^{\text{ème}}$ caractère via

```
char c = s[i] ;
```

Notez que les caractères sont des nombres codés sur 8 bits, donc entre 0 et 255. Vous pouvez donc facilement compter les caractères en initialisant un tableau de 256 zéros et en y accédant en utilisant le caractère. Par exemple :

```
tab['a'] = 12 ; //inscrire 12 dans la case du tableau correspondant au caractere a
```

Pour réaliser la fusion des arbres via la file a priorité, nous vous proposons une file a priorité dans les fichiers `huff_heap.[hc]pp`. Si vous vous sentez plus à l'aise avec une autre structure de données, vous n'êtes pas obligés de l'utiliser, mais nous vous le recommandons fortement. Pour que cette structure fonctionne, vous devez avoir codé correctement la fonction `huff_poids(HArbre* a)`. Cette fonction renvoie simplement le poids du sous-arbre, selon la façon dont vous l'avez stockée dans votre arbre.

## 5.3 Encodage et décodage

Une fois l'arbre de Huffman construit, vous pouvez attaquer l'encodage et le décodage d'une chaîne de caractères via cet arbre.

### 5.3.1 Décodage

Le décodage est l'opération la plus simple. Elle prend en paramètre l'adresse du nœud racine de l'arbre de Huffman ayant servi à encoder. Il prend également en paramètre le code à décoder, fourni sous la forme d'un tableau de bits via le type `std::vector<bool>`. Pour parcourir ce tableau, vous pouvez obtenir sa taille via

```
int taille = code.size() ;
```

et pour accéder au i$^{\text{ème}}$ élément, le vector se comporte comme un tableau :

```
bool bit = code[i] ;
```

Le dernier paramètre est la chaîne de caractères à remplir. Pour ajouter un caractère dans cette chaîne, vous pouvez utiliser

```
s.push_back('a') ; //ajout du caractere a en fin de chaine
```

L'algorithme pour le décodage consiste donc à utiliser les bits du code un par un dans l'ordre pour descendre dans l'arbre à droite ou à gauche. Dès qu'une feuille est atteinte, le caractère de cette feuille est ajouté à la chaîne de caractères de sortie, et le parcours de l'arbre redémarre de la racine.

### 5.3.2 Encodage simple mais coûteux

L'encodage est un peu plus compliqué. Nous vous proposons tout d'abord une version simple. La fonction d'encodage prend en paramètre l'adresse du nœud racine de l'arbre de Huffman à utiliser, la chaîne de caractères à encoder, et le tableau dynamique de bits (booléens) à remplir.

```
out.push_back(true) ; // ajout d'un 1
out.push_back(false) ; // ajout d'un 0
```

Nous vous proposons de commencer par écrire une fonction qui prend en paramètre un nœud de Huffman et un caractère, et qui renvoie un booléen indiquant si le caractère correspond à l'une des feuilles descendantes du nœud. De cette manière, depuis la racine, vous pouvez déterminer le premier bit du code d'un caractère en regardant s'il se trouve dans le sous-arbre de gauche ou de droite. Une fois le premier bit déterminé, vous pouvez descendre dans le sous-arbre correspondant et recommencer pour déterminer le second bit, jusqu'à vous trouver sur la feuille correspondant au caractère.

### 5.3.3 Encodage amélioré

L'algorithme précédent calcule de multiples fois la même chose en lançant de multiples recherches d'un caractère. De plus chaque recherche va potentiellement explorer tous les nœuds du sous-arbre sur lequel elle est lancée. Pour améliorer l'algorithme précédent, vous pouvez commencer par lister les feuilles de l'arbre, leurs caractères, et les code associés. Comme lors de la construction de l'arbre, vous pouvez vous aider d'un tableau de taille 256 pour associer des informations à des caractères.