# TP C#13 : Brainfuck

## 1 Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive de rendu respectant l'architecture suivante :

```
tpcs13-login_x.zip
    |– tpcs13-login_x/
        |– AUTHORS
        |– README
        |– src/
            |– Brainfuck.sln
            |– Brainfuck/
                |– Brainfuck.cs
                    |– Huffman.cs.cs
                    |– IDE.cs
                    |– Program.cs
                    |– Tree.cs
                    |– ...
                |– tout sauf bin/ et obj/
```

Vous devez, bien entendu, remplacer login_x par votre propre login. Avant de rendre, n'oubliez pas de vérifier :

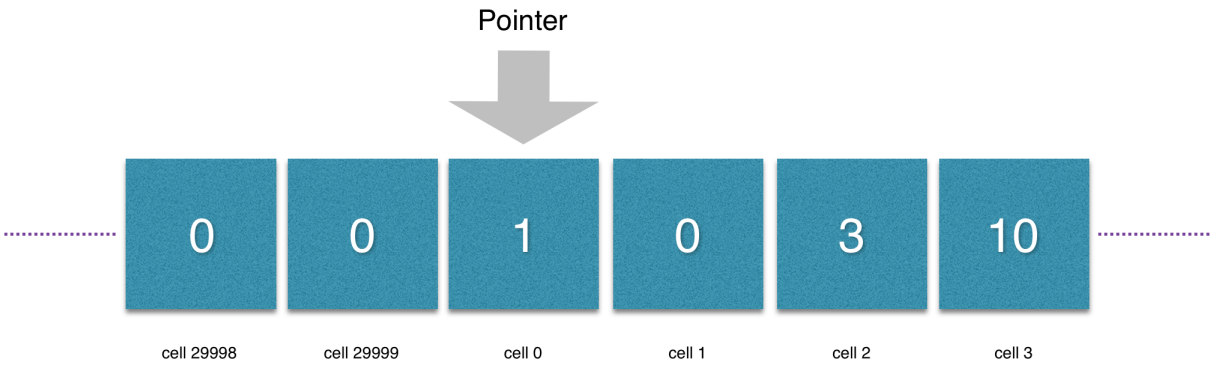
- Que le fichier AUTHORS est bien au format habituel (une *, un espace, votre login et un retour à la ligne, dans un fichier sans extension)
- Que vous avez bien supprimé les dossiers bin et obj
- **Que votre code compile**

# 2 Le Brainfuck

## 2.1 Généralités

Le Brainfuck est un langage de programmation exotique créé par Urban Müller en 1993. L'objectif de ce langage est d'être le plus simple possible mais en restant Turing-Complet et donc il est théoriquement possible d'écrire n'importe quel programme en Brainfuck.

Tout comme la machine de Turing le langage est composé d'un "ruban" ou d'un tableau stockant des octets. Il est possible de se déplacer d'une case à l'autre. Un pointeur permet de garder la position de la case courante du tableau.




Le tableau est généralement un tableau circulaire de 30000 cases.


## 2.2 Instructions

Le langage est composé de seulement 8 symboles différents, chaque symbole correspondant à une instruction. Voici la liste de ces instructions :

- **'>'** : incrémente le pointeur p
- **'<'** : décrémente le pointeur p
- **'+'** : incrémente la valeur de la case du tableau pointeé par le pointeur p
- **'-'** : décrémente la valeur de la case du tableau pointeé par le pointeur p
- **'.'** : affiche le caractère pointé par le pointeur p(valeur case = code ASCII)
- **','** : lit un caractère sur l'entrée standard et le stocke dans la case courante du tableau
- **'['** : saute à l'instruction après le ] correspondant si l'octet pointé est à 0.
- **']'** : retourne à l'instruction après le [ si l'octet pointé est différent de 0.

```
1 ++++++++++[>+++++++>++++++++++>+++>+<<<<-]
2 >++.>+.+++++++..+++.>++.<<+++++++++++++++.
3 >.+++.------.--------.>+.>.
```

Hello World en Brainfuck

## 2.3 Implémentation

Vous l'avez compris l'objectif de ce TP est de coder un interpréteur de Brainfuck. Le tableau sera un tableau d'**Int** de taille 30000. Le tableau devra être géré de façon circulaire.
Le pointeur sera implémenté sous forme d'un Int qui représentera l'index de la case pointée.
Les valeurs stockées devront être comprises entre **0** et **255** inclus. N'oubliez pas de gérer l'overflow (la valeur **256** passe à **0** et la valeur **-1** passe à **255**).


## 2.4 Exercices

Les méthodes à coder dans les exercices ci-dessous sont à coder dans la classe **Brainfuck**.

### 2.4.1 There are no facts, only interpretations

```
1 public static void interpret(string code);
```

La méthode **interpret** est une classe permettant d'interpréter un code Brainfuck donné en paramètre. Plusieurs etapes :

- Déclaration des variables selon les spécifications de l'implémentation définie plus haut.
- Initialisation du tableau.
- Pour chaque caractere réaliser l'instruction correspondante.

Protip : Pour la gestion des boucles vous pouvez utiliser une **Stack** (cf MSDN et Cours d'algo) pour stocker leurs positions dans le code.

### 2.4.2 Never trust a computer you can't throw out a window

```
1 public static string generate_code_from_text(string text);
```

Cette méthode renvoie le code Brainfuck permettant d'écrire la phrase contenue dans text.

Protip : Le code **"[-]"** permet de remettre la case actuelle à 0.

### 2.4.3 And one more thing

```
1 public static string shorten_code(string program);
```

**Shorten_code** doit renvoyer un code fonctionnel faisant la même chose que le code original mais avec moins de caractères. Vous pouvez utiliser toutes les techniques que vous voulez du moment que le code produit est plus court que le code original.


Pour la partie mandatory de cet exercice vous avez a repérer les longues séquences de + pour les remplacer par des multiplications (à l'aide de boucles).
Si d'autres techniques sont utilisées merci de le mentionner dans le Readme pour valider le Bonus.

# 3 Huffman

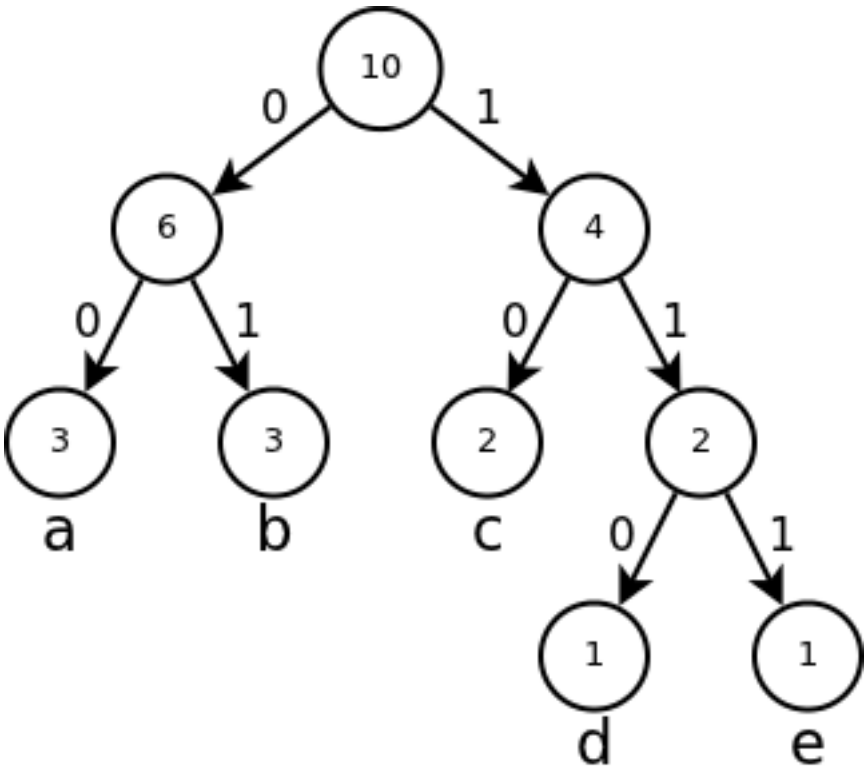
## 3.1 Généralités

Le codage de Huffman est un algorithme de compression de données sans perte élaboré par David Albert Huffman. Le principe est de créer un arbre binaire qui respecte deux règles :

— Chaque feuille est un symbole que l'on trouve dans le fichier à compresser.
— Un symbole dont le nombre d'occurrences est plus bas qu'un autre sera plus profond dans l'arbre.

Une fois l'arbre créé, chaque symbole peut se voir assigner un code binaire en parcourant l'arbre de la racine jusqu'à la feuille. Passer par un fils gauche équivaut à un 0 et par la droite à un 1. Comme on l'a vu, un symbole utilisé beaucoup de fois sera placé à une profondeur basse et donc aura un code binaire plus court.

Pour plus d'informations : `https://en.wikipedia.org/wiki/Huffman_coding`


## 3.2 Implémentation

Pour ce Tp nous allons compresser seulement du code Brainfuck, vous connaissez donc déjà quels symboles peuvent être présents.
Coder une version générale est un Bonus. Indiquez dans votre Readme si vous l'avez fait.
En premiers lieu il faut compter le nombre d'occurrences de chaque symbole et créer des nœuds qui contiennent à la fois le symbole et le nombre d'occurrences.
Ensuite nous avons deux implémentations possibles pour construire l'arbre :

**Première solution**

- Ajouter les nœuds dans une file de priorité où les basses occurrences sont prioritaires.
- Temps qu'il reste plus d'un nœud dans la file :
  - Défiler les deux nœuds avec la plus haute priorité (plus basses occurrences).
  - Créer un nœud avec un nombre d'occurrences égale à la somme des deux nœuds précédemment défilés. Ces derniers deviennent les fils du nouveau nœud.
  - Enfiler le nouveau nœud
- Le dernier nœud dans la file est la racine de l'arbre.

**Seconde solution**

- Créer deux files et ajouter les nœuds dans la première file dans l'ordre croissant du nombre d'occurrences.
- Tant qu'il y a plus d'un nœud dans les files.
  - Défiler les deux nœuds avec les plus basses occurrences en comparant les deux files.
  - Créer un nœud avec un nombre d'occurrences égale à la somme des deux nœuds précédemment défiler. Ces derniers deviennent les fils du nouveau nœud.
  - Enfiler le nouveau nœud dans la seconde file
- Le dernier nœud dans l'une des files est la racine de l'arbre.

Une fois l'arbre créé, on peut en déterminer une table mettant en relation le symbole avec son code binaire.
Il suffit ensuite de copier cette table dans le fichier texte compressé au début qu'on appellera la préface. Pour finir on code (en fonction de la table) chaque caractère.
La préface sert pour la décompression, en la lisant vous pourrez recréer la table de relation symbole/code et ainsi décoder le reste du fichier.


## 3.3 Exercices

Les methodes correspondants aux exercices ci-dessous sont à coder dans la classe **HuffmanCode**.

### 3.3.1 Constructeur

```
1 public HuffmanCode(string program);
```

Dans le constructeur vous devez juste appeler les méthodes pour générer l'arbre d'Huffman et pour coder la chaîne de caractères passés en paramètre en fonction de cet arbre.

### 3.3.2 You bred Huffman Tree !

```
1 private void generate_huffman_code(string program);
```

Cette méthode sert à construire l'arbre d'Huffman. Comme vu précédemment, il existe pour cela deux méthodes à votre disposition.

Protips : En plus des 8 symboles du Brainfuck, il peut être intéressant de rajouter un symbole présent une unique fois qui représente la fin du code pour faciliter le décodage.

### 3.3.3 Trust in me

```
1 private void generate_huffman_code_rec(Tree t, string path,
      Dictionary<char, string> encodings);
```

Une fois l'arbre créé, cette méthode fait un parcours profondeur récursif pour assigner à chaque symbole son code.
Rappels :

- Chaque feuille de l'arbre équivaut à un symbole.
- Le code binaire d'un symbole est donné en parcourant la branche de la racine jusqu'à la feuille correspondante.


— Un passage sur le fils gauche équivaut à un 0
— un passage sur le fils droit équivaut à un 1

### 3.3.4 No quote, just display please

```
1 public void print_code();
```

Affiche la table symbole/code dans la console.

### 3.3.5 I will be back

```
1 public string get_huffman_from_brainfuck();
```

Cette méthode va créer et retourner la chaîne de caractères codée, il y a plusieurs étapes à suivre :

— Créer la préface contenant la table des symboles. Plus cette préface est écrite de façon compressée mieux c'est. N'oubliez pas de trouver un moyen de savoir où se finit votre préface.
— Coder caractère par caractère la chaîne de caractères.
  — Dès que vous avez assez de bits (8 bits) pour créer un caractère, ajouter ce caractère à la chaîne de caractères codée
  — Si à la fin vous n'avez pas une suite de 8 bits, rajoutez des 0 pour finir.

### 3.3.6 I'm back

```
1 get_brainfuck_from_huffman(string huffman_code);
```

Cette méthode permet de décoder une chaîne de caractères précédemment codée pour retrouver du code Brainfuck. Les étapes à suivre sont :

— Recréer la table de relation symbole/code à l'aide de la préface.
— Créer une chaîne de caractères où les caractères sont remplacés par leur équivalent binaire.


— Lire cette chaîne de caractères de 0 et de 1, caractère par caractère en vérifiant si la séquence courante est dans la table.
  — **Si oui** : rajouter le caractère correspondant à la chaîne de caractères binaire et commencer une nouvelle séquence de bit.
  — **Si non** : ajouter un caractère à la séquence.
— Arrêter quand vous repérez le caractère unique de fin de code que vous aviez ajouté.

Il est évidemment possible de passer l'étape de transformation des caractères en leur équivalent binaire et de faire le traitement à la volée.