CODAGE DE HUFFMAN

Les structures de données

CODAGE DE HUFFMAN

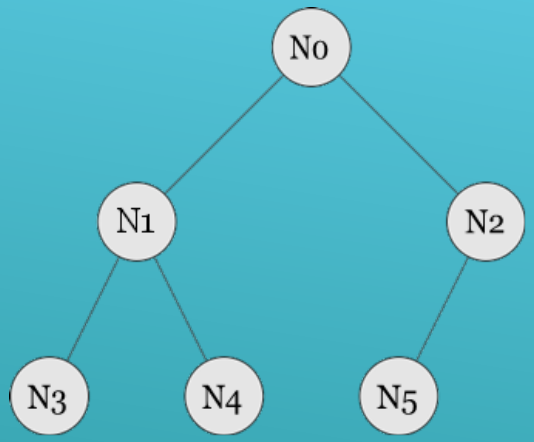
Le codage de Huffman est un algorithme de compression de données sans perte. Le codage de Huffman utilise un code à longueur variable pour représenter un symbole de la source (par exemple un caractère dans un fichier).

Le code est déterminé à partir d'une estimation des probabilités d'apparition des symboles de source, un code court étant associé aux symboles de source les plus fréquents.

Un code de Huffman est optimal au sens de la plus courte longueur pour un codage par symbole, et une distribution de probabilité connue. Des méthodes plus complexes réalisant une modélisation probabiliste de la source permettent d'obtenir de meilleurs ratios de compression.

Il a été inventé par David Albert Huffman, et publié en 1952.

David Albert Huffman
(1925 – 1999)

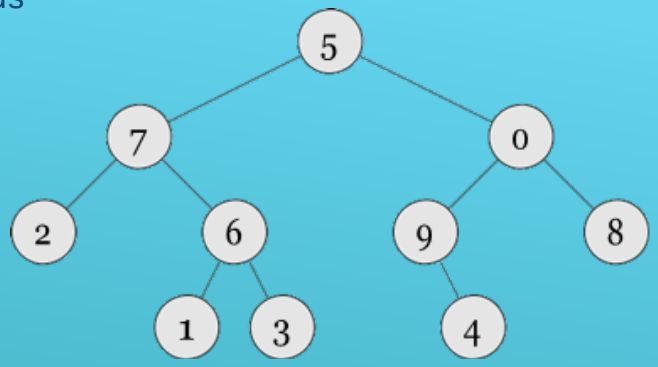


- N_0 est la racine de l'arbre.
- On appelle nœud interne un nœud possédant au moins un enfant. N_1, N_2 sont des nœuds internes.
- On appelle feuille un nœud ne possédant aucun enfant.
- N_3, N_4, N_5 sont des feuilles
- On appelle ascendant d'un nœud l'ensemble du parent et de ses ascendants.
- On appelle descendant d'un nœud l'ensemble des enfant et de leurs descendants.
- Un **arbre binaire** est un arbre où les nœuds ont au plus **deux enfants**. On appelle dans ce cas les **enfants gauche** et **enfants droit**.
- Le chemin de la racine à un nœud est la suite de nœuds passant de parent en enfant de la racine jusqu'au nœud (ces ascendants).
- La profondeur d'un nœud est le nombre de ces ascendants.
- La hauteur d'un arbre est la profondeur maximale des feuilles. On peut définir également la hauteur de façon récursive. La hauteur d'un arbre vide est nul. La hauteur d'un arbre non vide vaut 1+ la plus grande hauteur des sous arbres de la racine.

RAPPEL ARBRE BINAIRE

La plupart des algorithmes sur les arbres nécessitent de parcourir tous les nœuds. Plusieurs méthodes sont donc possibles pour explorer chaque nœud qui sont :

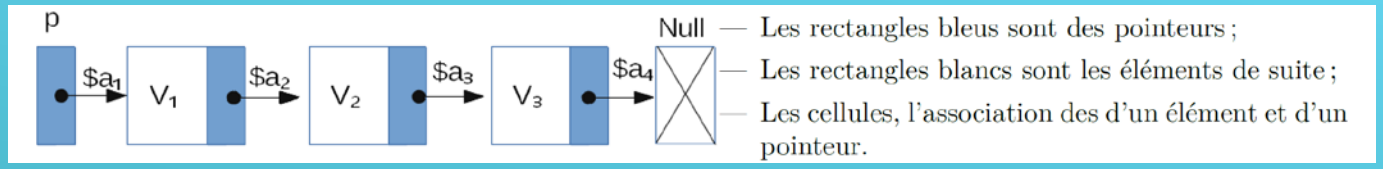
- le parcours en largeur
- le parcours en profondeur (récursif)
  - Préfixe
  - Infixe
  - Suffixe (postfixe)

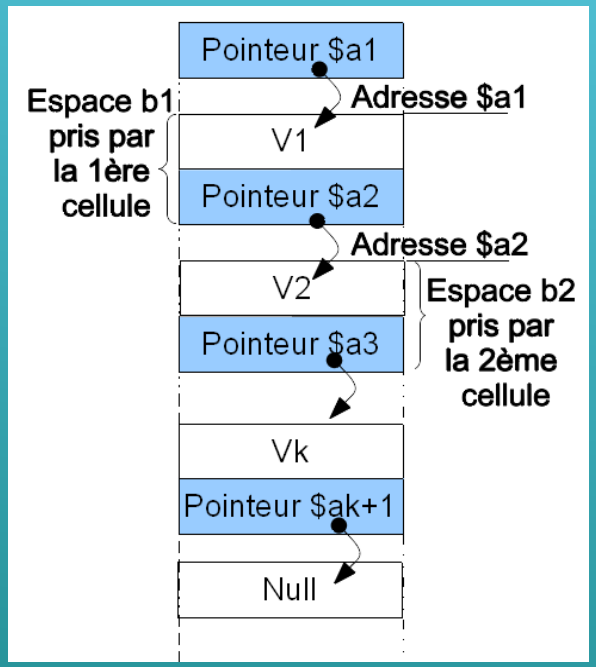


Ici :

- Parcours en largeur : 5 7 0 2 6 9 8 1 3 4 (pas d'intérêt particulier)
- Récursif préfixé : Racine -> Gauche -> Droite : 5 7 2 6 1 3 0 9 4 8
- Récursif infixé : Gauche -> Racine -> Droite : 2 7 1 6 3 5 9 4 0 8
- Récursif suffixé : Gauche -> Droite -> Racine : 2 1 3 6 7 4 9 8 0 5

ARBRES BINAIRES - PARCOURS





Pour définir une liste dynamique simplement chaînées, il suffit de définir l'entité cellule composée d'une tête contenant la valeur et d'une queue contenant le pointeur vers la prochaine cellule.

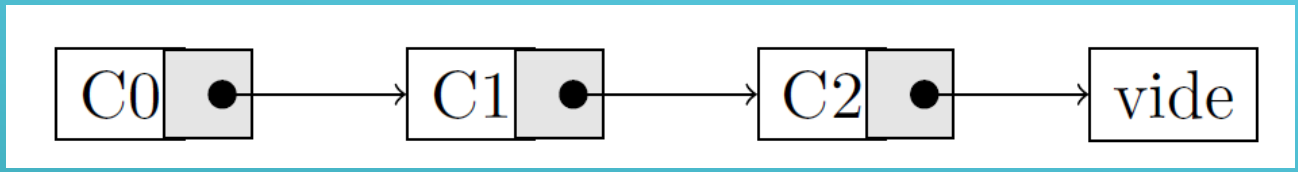
En python :

```python
class cellule :                     # Définition d'une classe d'objet

    def __init__(self,data):        # Méthode de construction de l'objet
        self.valeur = data          # Attribut de stockage

        self.pointeur = None        # Attribut ou champ qui contiendra l'adresse
                                    # de la cellule suivante


#  Exemple : on instancie 3 cellules contenant 5 , 8 , 9

a = cellule (5)

b = cellule (8)

c = cellule (9)

a.pointeur = b
b.pointeur = c


print(a)
print(a.valeur)
print(a.pointeur.valeur)
print(a.pointeur.pointeur.valeur)
```

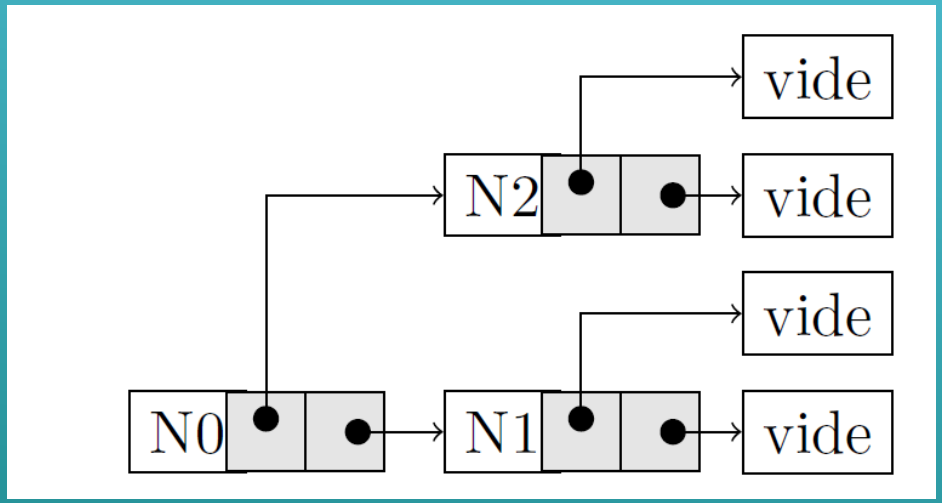
LISTES CHAINÉES (LDSC)

```
class cellule :                    # Définition d'une classe d'objet

    def __init__(self,data):       # Méthode de construction de l'objet
        self.valeur = data         # Attribut de stockage

        self.pointeur = None       # Attribut ou champ qui contiendra l'adresse
                                   # de la cellule suivante



#  Exemple : on instancie 3 cellules contenant 5 , 8 , 9

a = cellule (5)

b = cellule (8)

c = cellule (9)

a.pointeur = b
b.pointeur = c


print(a)
print(a.valeur)
print(a.pointeur.valeur)
print(a.pointeur.pointeur.valeur)
```

- Les arbres binaires peuvent être implantés facilement en machine. Il ressemble beaucoup aux listes chaînées à la différence qu'un nœud a deux successeurs tandis qu'une cellule de la chaîne n'en a qu'un seul.



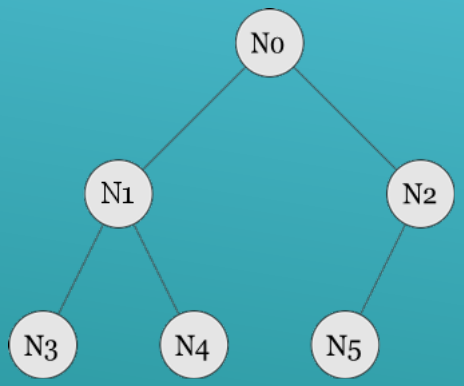
LDSC



Arbre binaire

STRUCTURE DE DONNÉES DES A.B.

```
class Noeud():
    def __init__(self,g,l,d):
        self.gauche = g
        self.Label = l
        self.droite = d
```




```
class Noeud():
    def __init__(self,g,l,d):
        self.gauche = g
        self.Label = l
        self.droite = d

# on commence par la canopée
Feuille1 = Noeud(None,'N3',None)
Feuille2 = Noeud(None,'N4',None)
Feuille3 = Noeud(None,'N5',None)


noeud1 = Noeud(Feuille1,'N1', Feuille2)
noeud2 = Noeud(Feuille3,'N2', None)

Racine = Noeud(noeud1,'N0', noeud2)

print(Racine.droite.gauche.Label)
```

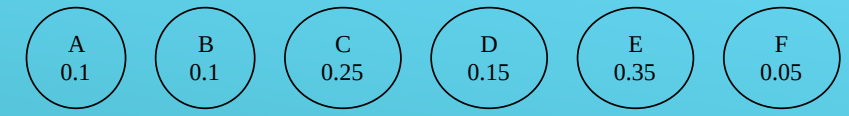
CODAGE

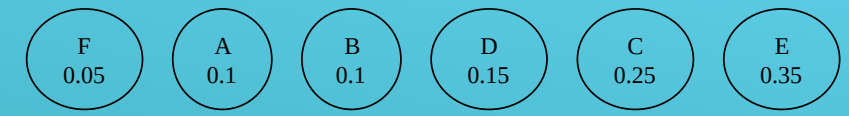
ALGORITHME

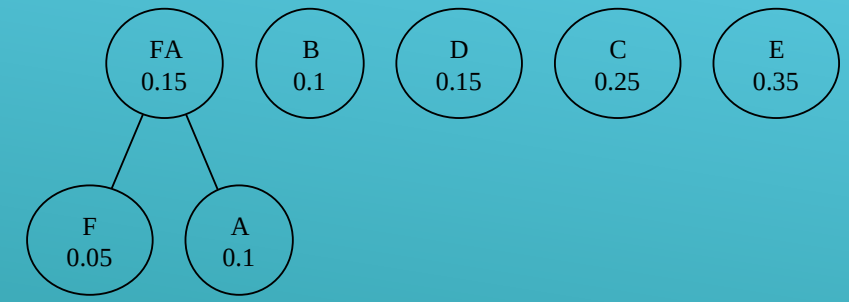
Compression d'Huffman

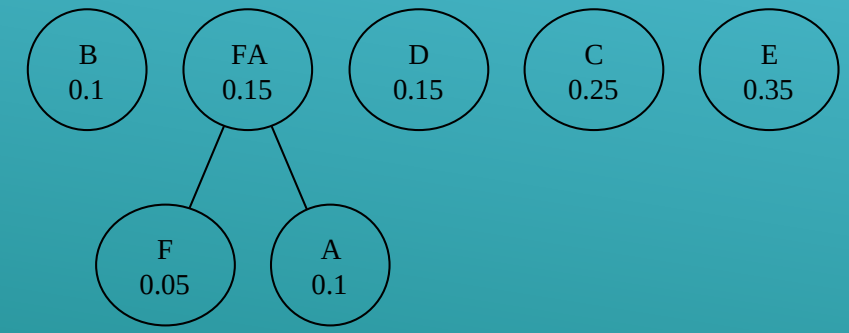
HUFFMAN - PRINCIPE

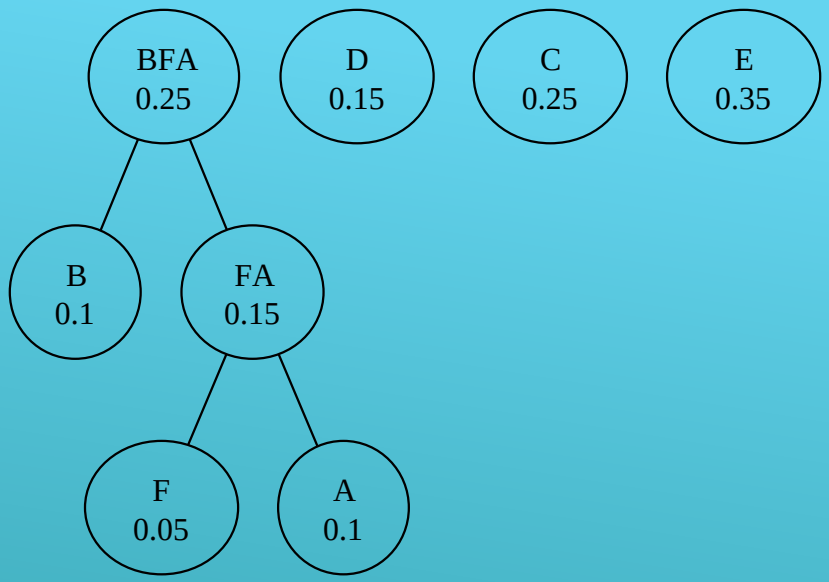
- Il s'agit de coder un texte avec le moins de bits possible. On associe à chaque caractère un mot binaire en attribuant les mots binaire les plus court aux caractères les plus fréquemment rencontrés.
- Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids égal à la fréquence d'occurrences.
- L'arbre est créé de la manière suivante, on associe chaque fois les deux nœuds de plus faibles poids, pour donner un nouveau nœud dont le poids équivaut à la somme des poids de ses enfant. On réitère ce processus jusqu'à n'avoir plus qu'un seul nœud : la racine. On associe ensuite par exemple le code 0 à chaque embranchement partant vers la gauche et le code 1 vers la droite.
- Pour obtenir le code binaire de chaque caractère, on remonte l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code un 0 ou un 1 selon la branche suivie.

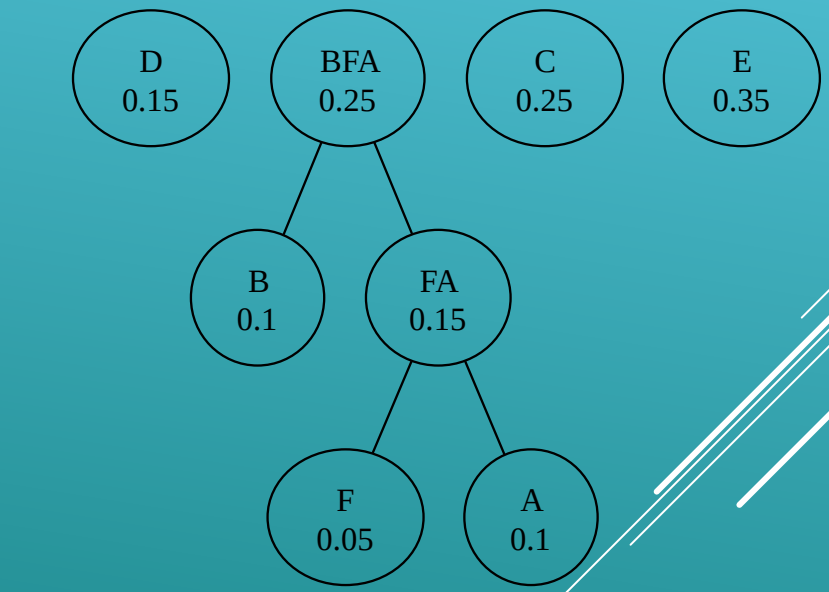

A 0.1
B 0.1
C 0.25
D 0.15
E 0.35
F 0.05


F 0.05
A 0.1
B 0.1
D 0.15
C 0.25
E 0.35

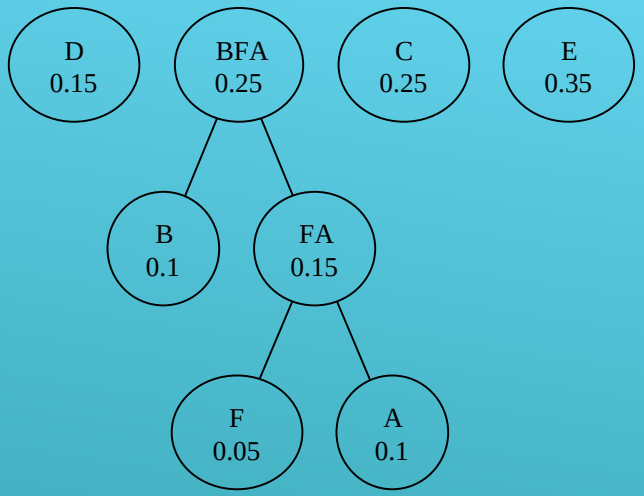

FA 0.15
B 0.1
D 0.15
C 0.25
E 0.35
F 0.05
A 0.1

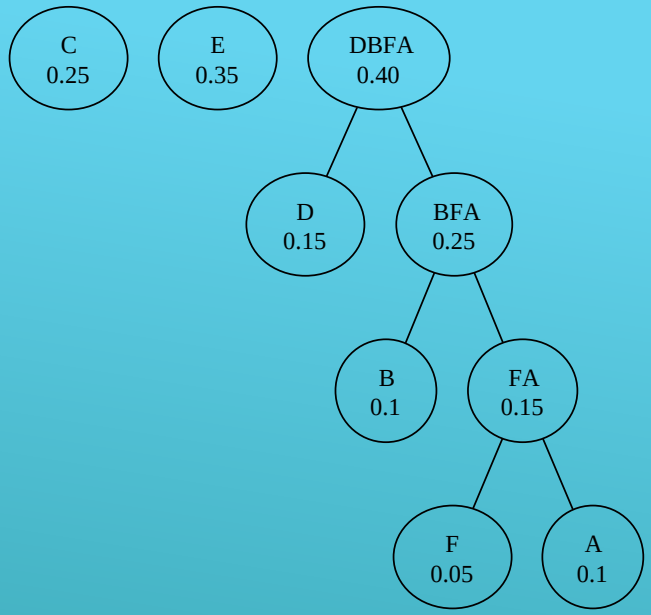

B 0.1
FA 0.15
D 0.15
C 0.25
E 0.35
F 0.05
A 0.1

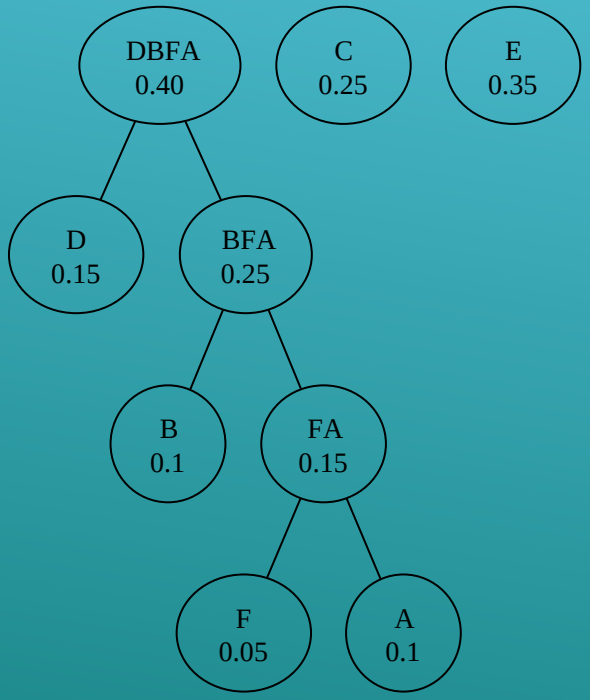

BFA 0.25
D 0.15
C 0.25
E 0.35
B 0.1
FA 0.15
F 0.05
A 0.1

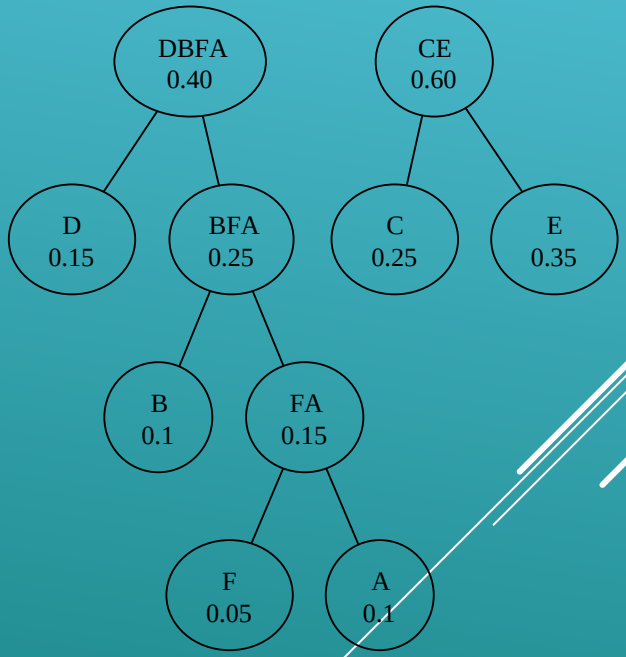

D 0.15
BFA 0.25
C 0.25
E 0.35
B 0.1
FA 0.15
F 0.05
A 0.1

HUFFMAN

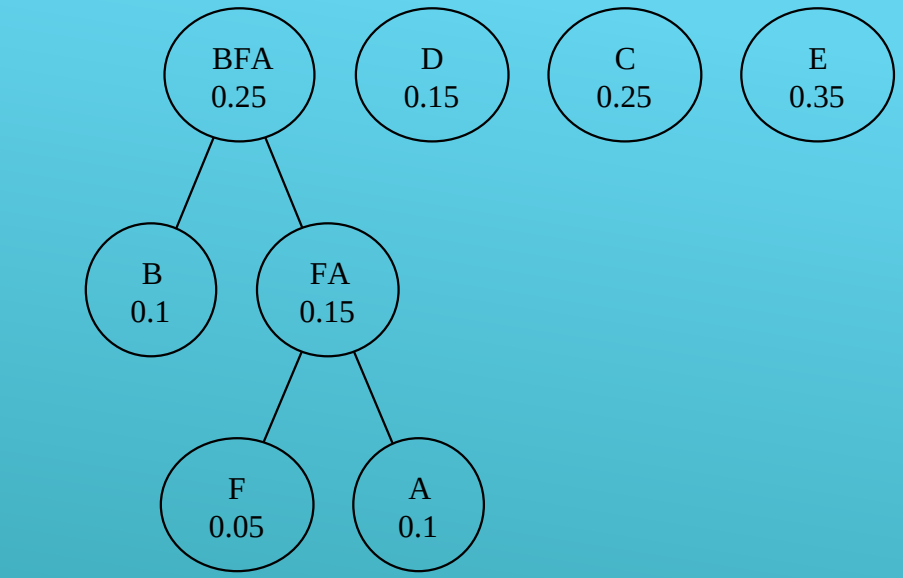

D
0.15
BFA
0.25
C
0.25
E
0.35
B
0.1
FA
0.15
F
0.05
A
0.1

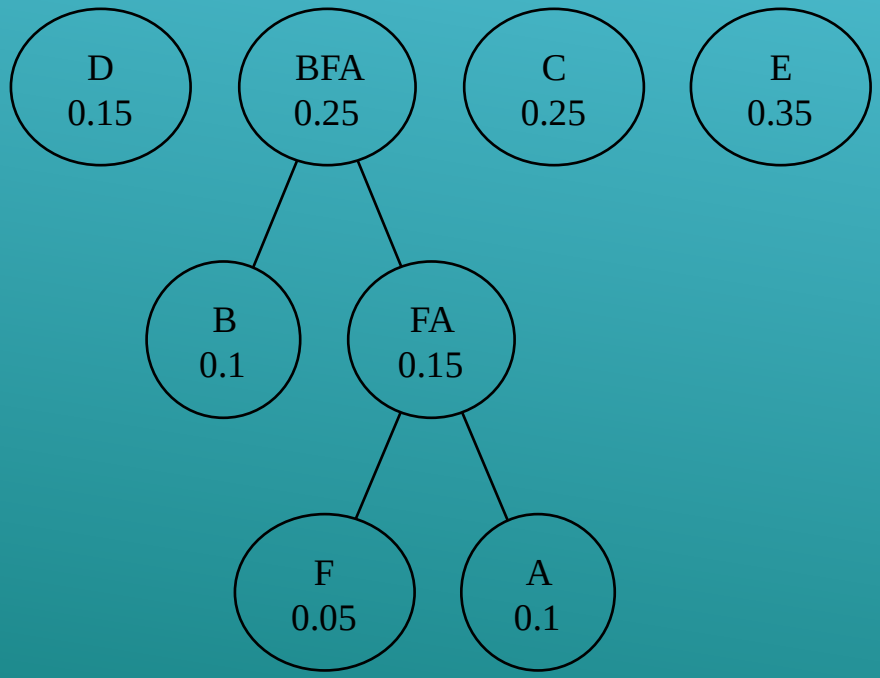

C
0.25
E
0.35
DBFA
0.40
D
0.15
BFA
0.25
B
0.1
FA
0.15
F
0.05
A
0.1


DBFA
0.40
C
0.25
E
0.35
D
0.15
BFA
0.25
B
0.1
FA
0.15
F
0.05
A
0.1

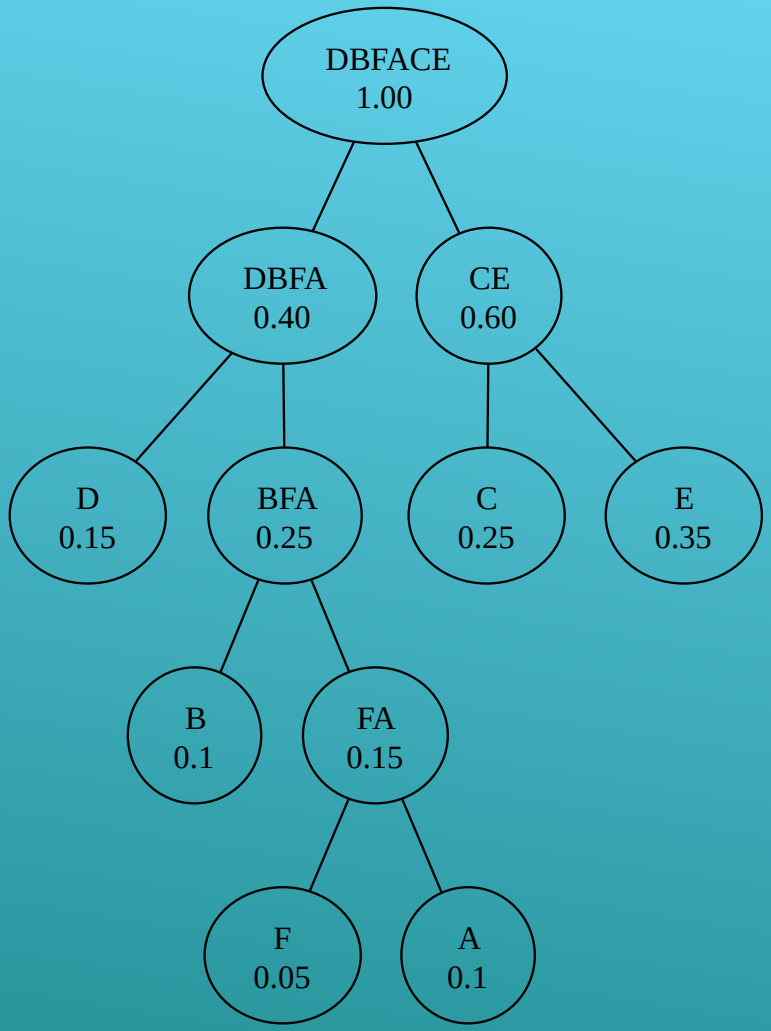

DBFA
0.40
CE
0.60
D
0.15
BFA
0.25
C
0.25
E
0.35
B
0.1
FA
0.15
F
0.05
A
0.1

HUFFMAN


BFA
0.25
D
0.15
C
0.25
E
0.35
B
0.1
FA
0.15
F
0.05
A
0.1


D
0.15
BFA
0.25
C
0.25
E
0.35
B
0.1
FA
0.15
F
0.05
A
0.1

HUFFMAN



symbole	*code*
A	0111
B	010
C	10
D	00
E	11
F	0110

HUFFMAN

```
def code (n,car):
    if n.gauche != None :
        if car in n.gauche.label:
            return '0'+code(n.gauche,car)
        else:
            return '1'+code(n.droite,car)
    else:
        return ''
```

```
def table_code (arb,f):
    tab={}
    for car in f:
        tab[car]=code(arb,car)
        print(car,' -> ',tab[car])
    return tab
```

TABLE DE CODAGE

```
### Prg principal
Texte = 'Lorem ipsum dolor sit amet, (

canope=feuilles(realise_dico_freq(Tex

liste_car=[]
for car in canope:
    liste_car.append(car.label)

arbre=cree_arbre(canope)

table = table_code(arbre,liste_car)

print(arbre.label)
print(arbre.frequence)
print (table)
```

PROGRAMME PRINCIPAL

- graphviz