Master : Science des Données, Génie Logiciel et Réseaux (SDGL&R)

Année universitaire : 2023-2024

Module : Structures de données avancées(S1)

Pr. S. EL KASMI

La compression de HUFFMAN

La **compression de Huffman** est une méthode de compression de données sans perte proposé par « David Huffman » en 1952. C'est un processus qui permet de compresser des données informatiques afin de libérer de la place dans la mémoire d'un ordinateur. Or tout fichier informatique (qu'il s'agisse d'un fichier texte, d'une image ou d'une musique …) est formé d'une suite de caractères. Chacun de ces caractères étant lui-même code par une suite de 0 et de 1. L'idée du codage de Huffman est de repérer les caractères les plus fréquents et de leur attribuer des codes courts (c'est-a-dire nécessitant moins de 0 et de 1) alors que les caractères les moins fréquents auront des codes longs. Pour déterminer le code de chaque caractère on utilise un arbre binaire. Cet arbre est également utilisé pour le décodage.

Dans cette partie, on s'intéressera à appliquer la compression de Huffman pour compresser une liste de nombres entiers, contenant des éléments en répétition.

Exemple : `L = [12,29,55,29,31,8,12,46,29,8,12,29,31,29,8,29,8]`

***NB** : Dans toute cette partie, on suppose que la liste L contient au moins deux éléments différents.*

II. 3- Liste des fréquences :

La première étape de la méthode de compression de Huffman consiste à compter le nombre d'occurrences de chaque élément de la liste des entiers.

Écrire une fonction : **`frequences(L)`**, qui reçoit en paramètre une liste d'entiers **L**, et qui retourne une liste de tuples : Chaque tuple est composé d'un élément de L et de son occurrence (répétition) dans L. Les premiers éléments des tuples de la liste des fréquences doivent être tous différents (sans doublon).

Exemple : L= [12, 29, 46, 29, 31, 8, 12, 55, 29, 8, 12, 29, 31, 29, 8, 29, 8]

La fonction **frequences (L)** retourne la liste F = [(**8**, 4), (**12**, 3), (**46**, 1), (**55**, 1), (**29**, 6), (**31**, 2)]

II. 4- Tri de la liste des fréquences :

La liste F des fréquences, des différents éléments de la liste des entiers, étant créée. L'étape suivante est de trier cette liste F dans l'ordre croissant des occurrences (les 2èmes éléments des tuples).

Écrire la fonction: **`tri(F)`**, qui reçoit en paramètre la liste F des fréquences, et qui trie les éléments de la liste F dans l'ordre croissant des occurrences. *Cette fonction doit être de complexité quasi-linéaire O(nlog(n)). Ne pas utiliser ni la fonction sorted( ), ni la méthode sort( ) de Python.*

Exemple : F = [(12, 3), (29, 6), (55, 1), (31, 2), (8, 4), (46, 1)]

La fonction **tri (F)** retourne la liste : [(46, **1**), (55, **1**), (31, **2**), (12, **3**), (8, **4**), (29, **6**)]

II. 5- Création de l'arbre binaire de HUFFMAN :

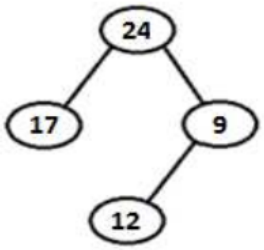
Pour représenter un arbre binaire, on va utiliser une liste composée de trois éléments :

- ✓ Chaque nœud est représenté par la liste : **[Père , [Fils gauche] , [Fils droit]]**
- ✓ Chaque feuille est représentée par la liste : **[Feuille , [ ] , [ ]]**

Exemple :

L'arbre binaire suivant, sera représenté par la liste :

```
A = [24,[17,[],[]],[9,[12,[],[]],[]]]
```



L'arbre binaire de Huffman est crée à partir de la liste des fréquences, selon le procédé suivant :

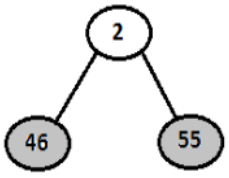
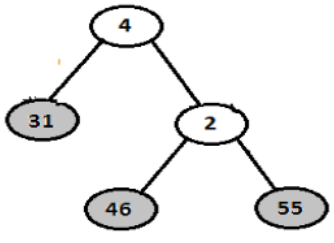
- ➢ *Trier la liste des fréquences dans l'ordre croissant des occurrences ;*
- ➢ *Tant que la liste des fréquences contient au moins deux éléments, on répète les opérations :*
  - *Retirer les deux premiers éléments dont les poids ($2^{ème}$ élément de chaque tuple) sont les plus faibles, et créer un tuple dont le poids est la somme des poids de ces deux éléments ;*
  - *Insérer le tuple obtenu dans la liste des fréquences, de façon à ce qu'elle reste croissante.*

Les feuilles de l'arbre binaire obtenu sont les différents éléments de la liste à compresser.

Exemple :

La liste F des fréquences suivante, est triée dans l'ordre croissant des occurrences.

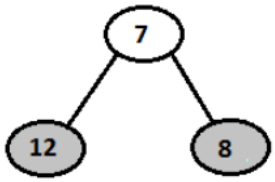
F = [(46, 1), (55, 1), (31, 2), (12, 3), (8, 4), (29, 6)]

$1^{ère}$ étape F = [**(46, 1), (55, 1),** (31, 2), (12, 3), (8, 4), (29, 6)] Retirer : **(46, 1)** et **(55, 1)** Insérer le tuple **(A , 2)** dans la liste F La liste F triée devient : F = [(31, 2), **(A , 2)**, (12, 3), (8, 4), (29, 6)]	**A** 
$2^{ème}$ étape F = [**(31 , 2), (A , 2),** (12, 3), (8, 4), (29, 6)] Retirer : **(31 , 2)** et **(A , 2)** Insérer le tuple **(A , 4)** dans la liste F La liste F triée devient : F = [(12, 3), (8, 4), **(A , 4)**, (29, 6)]	**A** 

3ème étape

F = [**(12, 3), (8, 4)**, (A , 4), (29, 6)]
Retirer : **(12 , 3)** et **(8 , 4)**
Insérer le tuple **(B , 7)** dans la liste F
La liste Ftriée devient :
F = [(A, 4), (29, 6), **(B , 7)**]

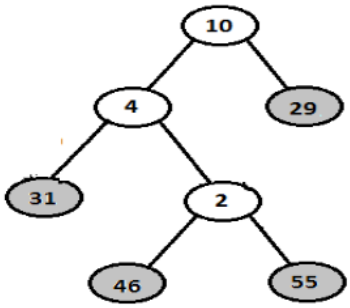
B



4ème étape

F = [**(A, 4), (29, 6)**, (B , 7)]
Retirer : **(A , 4)** et **(29 , 6)**
Insérer le tuple **(A , 10)** dans la liste F
La liste F triée devient :
F = [(B , 7), **(A , 10)**]

A

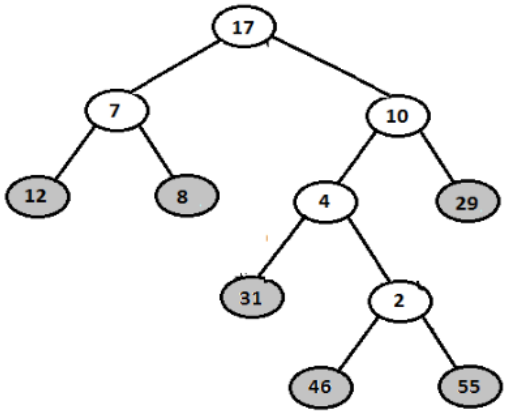


5ème étape

F = [**(B , 7), (A , 10)**]
Retirer : **(B , 7)** et **(A , 10)**
Insérer le tuple **(A , 17)** dans la liste F
La liste F triée devient :
F = **[(A , 17)]**

La liste F ne compte plus qu'un seul élément, le procédé est arrêté.

A



II. 5- a) Écrire la fonction de complexité linéaire : **`insere(F,T)`**, qui reçoit en paramètre la liste **F** des fréquences triée dans l'ordre croissant des occurrences, et un tuple **T**, de même nature que les éléments de **F**. La fonction insère le tuple **T** dans **F** de façon à ce que la liste **F** reste triée dans l'ordre croissant des occurrences.

Exemple : F = [(46, **1**), (55, **1**), (31, **2**), (12, **3**), (8, **4**), (29, **6**)] et T = **(17, 5).**

Après l'appel de la fonction : **insere (F, T)**, la liste F devient :

F = [(46, **1**), (55, **1**), (31, **2**), (12, **3**), (8, **4**), (17, **5**), (29, **6**)]

II. 5- b) Écrire la fonction : **`arbre_Huffman(F)`**, qui reçoit en paramètre la liste des fréquences **F**, composée des tuples formés des différents éléments de la liste à compresser, et de leurs occurrences. La fonction retourne un arbre binaire de Huffman, crée selon le principe décrit ci-dessus.

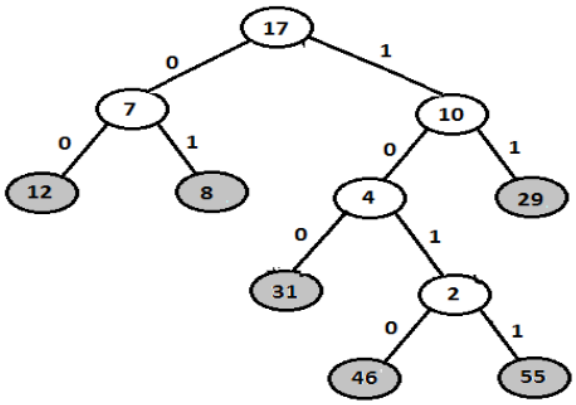
II. 6- Construction du dictionnaire des codes de HUFFMAN :

On suppose que l'arbre binaire de Huffman est crée selon le principe de la question précédente. L'étape suivante est de construire un dictionnaire où les clés sont les différents éléments de la liste à compresser (les feuilles de l'arbre de Huffman), et les valeurs sont les codes binaires de Huffman correspondants.

Parcours de l'arbre binaire de HUFFMAN :

On associe le code **0** à chaque branche de gauche et le code **1** à chaque branche de droite.

Pour obtenir le code binaire de chaque feuille, on parcourt l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code **0** ou **1** selon la branche suivie :



Ainsi, en parcourant l'arbre, on obtient les codes suivants :

Feuilles	*Codes*
12	**"00"**
8	**"01"**
31	**"100"**
46	**"1010"**
55	**"1011"**
29	**"11"**

Écrire la fonction : **`codes_Huffman(A, code, codesH)`** , qui reçoit trois paramètres :

- **A** : est un arbre binaire de Huffman, crée selon le principe de la question précédente.
- **code** : est une chaine de caractère initialisée par la chaine vide.
- **codesH** : est un dictionnaire initialisé par le dictionnaire vide

La fonction effectue un parcours de l'arbre, en ajoutant dans le dictionnaire **codesH**, les feuilles de l'arbre comme clés et les codes binaires correspondants comme valeurs des clés.

Exemple :

codesH = { } *# codesH est un dictionnaire vide.*

code = " " *# code est une chaine de caractères vide.*

Après l'appel de la fonction : **codes_Huffman (Arbre, code, codesH)**, Le contenu du dictionnaire codesH est :

{ 55 : '1011' , 8 : '01' , 12 : '00' , 29 : '11' , 46 : '1010' , 31 : '100' }

II. 7- Compression de la liste des entiers :

Le dictionnaire des codes binaires contient les différents éléments de la liste à compresser, et leurs codes Huffman binaires correspondants. On peut maintenant procéder à la compression de la liste des entiers.

Écrire la fonction : **`compresse(L,codesH)`**, qui reçoit en paramètres la liste **L** des entiers, et le dictionnaire **codesH** des codes binaires Huffman. La fonction retourne une chaine de caractères contenant la concaténation des codes binaires Huffman correspondants à chaque élément de L.

Exemple :

L= [12 , 29 , 46 , 29 , 31 , 8 , 12 , 55 , 29 , 8, 12, 29, 31, 29, 8, 29, 8]

codesH = { 55: '1011' , 8: '01' , 12: '00' , 29: '11' , 46: '1010' , 31: '100' }

La fonction **compresse (L, codesH)** retourne la chaine :

"00111010111000100101111010011100110111101"

II. 8- Décompression de la chaine de caractères binaires :

L'opération de décompression consiste à retrouver la liste initiale des nombres entiers, à partir de la chaine binaire et du dictionnaire des codes.

Écrire la fonction : **`decompresse(codesH, B)`**, qui reçoit en paramètres le dictionnaire **codesH** des codes de Huffman, et la chaine de caractères binaires **B**. La fonction construit et retourne la liste initiale.

Exemple :

B = "00111010111000100101111010011100110111101"

codesH = { 55: '1011', 8: '01', 12: '00', 29: '11', 46: '1010', 31: '100' }

La fonction **decompresse (codesH , B)** retourne la liste initiale :

[12, 29, 46, 29, 31, 8, 12, 55, 29 , 8, 12, 29, 31, 29, 8, 29, 8]