Université de Carthage Institut Préparatoire aux Études d'Ingénieur de Nabeul Département : Mathématiques	IPEIN المعهد التحضيري للدراسات الهندسية بنابل Institut Préparatoire aux Etudes d'Ingénieurs de Nabeul	Année universitaire : 2023/2024 Filière : SM/SP/ST Niveau d'étude : 2ème année Semestre : 2 Nombre de pages : 8 pages Date : 27/04/2024 Durée : 2h00

Concours Blanc Informatique

Problème 1.

Partie 1.

On s'intéresse des mesures de niveau de la surface libre de la mer effectuées par des bouées (dispositifs flottants). Ces bouées contiennent un ensemble de capteurs incluant un accéléromètre vertical qui fournit, après un traitement approprié, des mesures à étudier.

Les mesures réalisées à bord des bouées sont envoyées par liaison radio à une station à terre où elles sont enregistrées, contrôlées et diffusées pour servir à des études scientifiques.

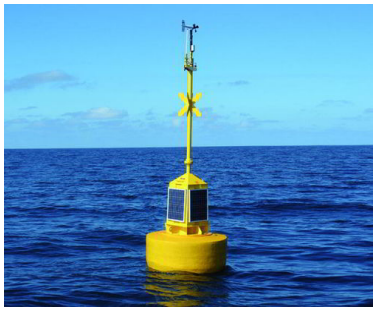


FIGURE 1 – Image de bouée en haute mer

Ces données sont enregistrées dans une base de données nommée "OceanData", laquelle contient les quatre tables suivantes :

1. **Bouee** (idBouee, nomSite, localisation, typeCapteur, frequence)
   Avec :
   - **idBouee** : le numéro d'identification de la bouée (clé primaire, entier)
   - **nomSite** : le nom du site ou est situé la bouée (chaîne de caractères)
   - **localisation** : le nom de la mer ou de l'océan (chaîne de caractères)
   - **typeCapteur** : le type du capteur (chaîne de caractères)
   - **frequence** : la fréquence d'échantillonnage (en Hz) du capteur (flottant)

IdBouee	nomSite	localisation	typeCapteur	frequence
831	Porquerolles	Mediterranee	Datawell non directionnelle	2
291	Les pierres noires	Mer d'iroise	Datawell directionnelle	1.28
...	...	...	...	...

TABLE 1 – Extrait de la table Bouee

2. **Campagne** (idCampagne,#idBouee, debutCampagne, finCampagne)
   Avec :
   - **idCampagne** : le numéro d'identification de la campagne d'enregistrement (clé primaire, entier)
   - **idBouee** : le numéro d'identification de la bouée (entier), clé étrangère, fait référence à la colonne `idBouee` de la table `Bouee`.
   - **debutCampagne** : la date de début de la compagne au format 'aaaa/mm/jj' (type Date)
   - **finCampagne** : la date de fin de la compagne au format 'aaaa/mm/jj' (type Date)

idCampagne	idBouee	debutCampagne	finCampagne
8301	831	2021/01/01	2021/05/15
2911	291	2023/06/01	2023/11/18
...	...	...	...

TABLE 2 – Extrait de la table Campagne

3. **Mesures** (#idBouee, Date, Hauteur)
   La table Mesures enregistre toutes les mesures envoyées par chaque bouée.
   - **idBouee** : le numéro d'identification de la bouée (entier), clé étrangère, fait référence à la colonne `idBouee` de la table `Bouee`.
   - **Date** : la date de prise de la mesure au format ***TimeStamp***, exprimée en secondes (réel).
   - **Hauteur** : hauteur du niveau de la mer, exprimée en mètres (réel).

   Le *Timestamp* est un type numérique utilisée pour représenter une date et une heure spécifiques, généralement exprimée en secondes depuis un point de référence fixe dans le temps (1er janvier 1970). Exemple de timestamp de "2014-04-27" est 1713518575.

idBouee	Date	Hauteur
831	1713518575	5.3
291	1713344521	2.1
...	...	...

TABLE 3 – Extrait de la table Mesures

4. **Tempete** (idTempete, #idBouee, debutTempete, finTempete, Hmax)
   La table Tempete enregistre la hauteur maximale du niveau de la surface de la mer pour chaque tempête.
   - **idTempete** : le numéro d'identification de la tempête (clé primaire, entier)
   - **idBouee** : le numéro d'identification de la bouée ayant mesuré cet épisode (entier), clé étrangère, fait référence à la colonne `idBouee` de la table `Bouee`.
   - **debutTempete** : la date de début de la tempête au format 'aaaa/mm/jj' (type Date)
   - **finTempete** : la date de fin de la tempête au format 'aaaa/mm/jj' (type Date)
   - **Hmax** : la hauteur maximale du niveau de la surface de la mer, exprimée en mètres (nombre décimal flottant), enregistrée entre les dates `debutTempete` et `finTempete`.

idTempete	idBouee	debutTempete	finTempete	Hmax
083010	831	2021/02/07	2021/02/09	5.3
029012	291	2023/08/16	2023/08/18	8.5
...	...	...	...	...

TABLE 4 – Extrait de la table Tempete

Indications :
Les différentes dates sont mémorisées comme des chaînes de caractères au format 'aaaa/mm/jj', ce qui permet de les comparer chronologiquement avec les symboles usuels (<, =, ...).

Questions :

Q1. Écrire une requête SQL qui fournit le nombre de bouées du site de Porquerolles.

Q2. Écrire une requête SQL qui fournit la liste des identifiants des tempêtes de Méditerranée qui se sont terminées avant le 1er janvier 2023 (strictement).

Q3. Écrire une requête SQL qui fournit la liste des noms des sites ayant connu au moins une tempête commencée en 2023.

Q4. Écrire une requête SQL qui fournit la liste des identifiants de campagne, de l'identifiant de la bouée utilisée et le nombre de tempêtes relevées au cours de chaque campagne de 2023 (le début et la fin ont eu lieu en 2023).

Q5. Écrire une requête SQL qui fournit la liste des identifiants des bouées n'ayant pas relevé de tempête en 2023 (le début et la fin ont eu lieu en 2023).

Partie 2.
Pendant la période d'enregistrement des mesures du niveau de la surface de la mer, des incidents peuvent survenir où certaines bouées cessent de fonctionner, entraînant un manque d'informations pendant la période de réparation.

Pour pallier cette situation, nous proposons de simuler ces mesures manquantes en utilisant les données enregistrées avant et après l'incident.

Plus précisément, nous envisageons d'appliquer le principe de l'interpolation de Lagrange sur les mesures enregistrées pendant une période donnée, définie par deux dates, `date_debut` et `date_fin` (représentant respectivement le début et la fin de la campagne).

Indications :
Pour convertir une date du format "aaaa-mm-jj" en valeur de type Timestamp, nous supposons avoir une fonction nommée `convert_to_timestamp(date_str)` qui retourne le Timestamp en secondes correspondant à la date spécifiée sous forme de chaîne de caractères, et inversement `convert_to_date(timestamp)` retourne la date correspondate au format "aaaa-mm-jj".
Exemples :
`convert_to_timestamp("2014-04-27")` renvoie le Timestamp correspondant : 1713518575
`convert_to_date(1713518575)` renvoie la date correspondante : "2014-04-27"

Q6. Écrivez une fonction Python nommée `get_Mesures(idBouee, date_debut, date_fin)` prenant comme paramètres l'identifiant de la bouée, la date de début et la date de fin de récupération de mesures, et qui, à partir de la base de données, retourne un tableau contenant les mesures enregistrées ainsi que leurs dates correspondantes (au format timestamp) par cette bouée entre les dates spécifiées.
Chaque élément du tableau retourné est un `tuple` sous cette forme `(date, hauteur)`.
Exemple d'utilisation :

```
date_debut = "2021-03-04"; date_fin = "2021-06-30" ; idBouee = 566
tab_mesures = get_Mesures(idBouee, date_debut, date_fin)
#tab_mesures = [(1617097290, 5.3), (1620639690, 8.5), ...]
```

On dispose d'un tableau de mesures contenant les couples (date, hauteur) et on veut connaître la fonction f qui lie ces points $(x_i : date, y_i : hauteur)$.
Nous allons ***interpoler*** cette fonction f, c'est-à-dire l'approcher par un ***polynôme Lagrange***, en s'arrangeant pour que la courbe représentative de ce polynôme passe par tous les points (x_i, y_i) de notre tableau.

La formule mathématique de l'interpolation de Lagrange :

$$P(x) = \sum_{i=0}^{n} Y_i \cdot L_i(x) \qquad avec \qquad L_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

Où :

- P : polynôme d'interpolation aux points x_i pour les mesures y_i ;
- x_i : nœuds ou points d'interpolations ;
- $y_i = f(x_i)$ représentent les valeurs interpolées ;

Q7. Écrivez une fonction Python nommée `interpolation_lagrange(X, Y, x)` qui permet d'implémenter la méthode de Lagrange et retourne une estimation du polynôme de lagrange au point x donné.
Cette fonction prend en paramètres

- Le vecteur $X = (x_0, x_1, ...x_n))$ formé des points d'interpolation ;
- Le vecteur $Y = (y_0, y_1, ...y_n))$ formé des valeurs d'interpolations ;
- Le point réel x auquel le polynôme se calcule ;

et retourne une estimation au point x.

Q8. Écrivez une fonction Python nommée `simulation_hauteur` prenant comme paramètres

- le tableau `tab_dates` contenant des dates au format Timestamp.
- le tableau de mesures de Hauteurs `tab_hauteurs`.
- une date `d_estim` (au format Timestamp) pour laquelle nous voulons estimer la valeur de la hauteur.

Cette fonction devrait retourner l'estimation de la hauteur de la surface de la mer à la date `d_estim` en utilisant l'interpolation de Lagrange.
Exemple d'utilisation :

```
d_estim = 1617200000  # Date pour laquelle nous voulons estimer la hauteur
# Appel de la fonction interpolation_lagrange
estimation = interpolation_lagrange(tab_dates, tab_hauteurs, d_estim)
# Affichage de l'estimation
print("Estimation de la hauteur à la date {} : {:.2f}".format(d_estim, estimation))
# Exemple de résultat de la hauteur estimée à la date 1617200000 : 5.30 mètres
```

Q9. Écrivez un script Python qui

1. Crée un tableau ***tab_x*** représentant l'intervalle des abscisses des estimations, et contenant les dates variant entre et `date_debut_campagne = 1617097290` et `date_fin_campagne = 1617356490` avec un pas égal à 1 seconde.
2. Applique l'interpolation de Lagrange pour estimer les valeurs des hauteurs de la surface de la mer pour chaque date du tableau ***tab_x*** et les enregistre dans un tableau ***tab_y***.

Q10. Écrivez un script Python qui à partir des données suivantes :

1. Identifiant d'une bouée `idBouee = 455`
2. Date début d'une campagne `date_debut = 1617097290`
3. Date fin d'une campagne `date_fin = 1617356490`

applique l'interpolation de Lagrange et génère sous forme d'un graphe une simulation de la variation de niveau de la mer pour toute la période la compagne.

Problème 2. Le Codage Huffman

Le codage Huffman est une technique utilisée pour coder et compresser les données brutes des fichiers numériques.

Soit un texte quelconque. Si nous utilisons le codage ASCII étendu, à 256 caractères, chaque caractère du texte est codé sur un octet. Donc un texte de 1000 caractères nécessite 1000 octets soit 8000 bits.
Le principe du codage de Huffman est le suivant : plutôt que coder chaque caractère sur 8 bits, on utilise moins de bits pour les caractères les plus fréquents et plus de bits pour les caractères les moins fréquents.

Considérons un exemple simple :
Un texte ne contient que les quatre caractères 'A', 'B', 'E' et 'R'. Le 'E' est présent 500 fois, le 'R' 300 fois, le 'A' 150 fois et le 'B' 50 fois.
Le codage Huffman génère des codes à longueur variable pour représenter les symboles (la génération est détaillée ci-dessous).
Finalement le codage est le suivant 'A' => 110, 'B' => 111, 'E' => 0, 'R' => 10
Le résultat compressé du mot "ARBRE" pourrait être représenté par la séquence de bits suivante `0b11010111100`.

Pour le texte de 1000 caractères, nous obtenons en nombre de bits : 500 x 1 bits pour les 'E', 300 x 2 bits pour les 'R', 150 x 3 bits pour les 'B', et 50 x 3 bits pour les 'B'. Le total est donc de 1700 bits au lieu des 8000 avec un codage sur 1 octet pour chaque caractère. Le gain est très important sur cet exemple.

Le taux de compression est donné par la relation suivante :

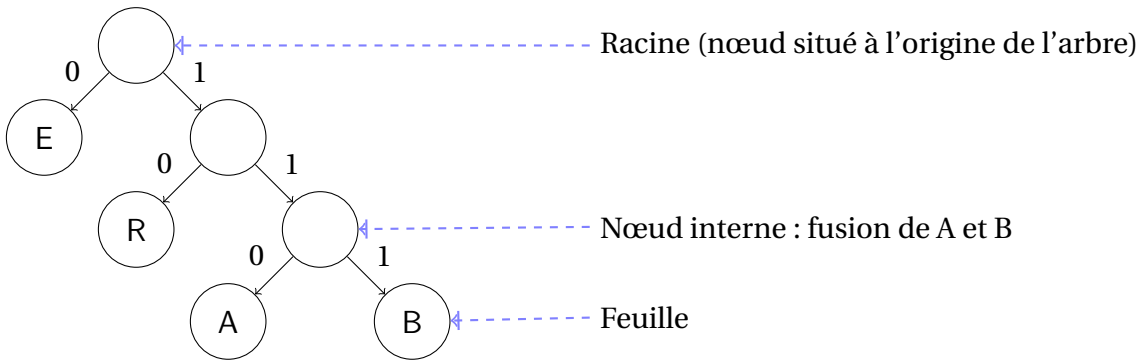
$$taux\_compression = \frac{taille\_initiale - taille\_compressée}{taille\_initiale} \times 100$$

Dans notre exemple il est d'environ 79 %

L'application du principe du codage de Huffman, repose sur la construction d'un ***arbre binaire*** où chaque caractère représente une **feuille**.

Un arbre binaire est une structure de données hiérarchique composée de nœuds. Chaque nœud peut avoir au plus deux enfants : un enfant gauche et un enfant droit. De manière récursive, chaque nœud peut être considéré comme un arbre en lui-même. Un nœud sans enfant est appelé une ***feuille*** ou un ***arbre élémentaire***.

Le *code* d'un caractère est donnée par le chemin suivi depuis la *racine* jusqu'à la *feuille* où il est situé, en associant un **0 à l'enfant gauche** et un **1 à l'enfant droit**.
D'après l'arbre de Huffman suivant, Le code 'E' est 0, le code 'R' est 10, le code 'A' est 110 et le code de 'B' est 111.



Pour obtenir les codes de chacun des caractères nous allons modéliser les nœuds de l'arbre Huffman par des objets de type **Arbre** (à définir).

La classe **Arbre** est caractérisée par :

- Attributs d'instances :
  — **char** : représente le caractère à coder si l'instance est une feuille, ou `None` sinon.
  — **NœudGauche** : représente l'enfant gauche ou `None` (si l'instance est une feuille).
  — **NœudDroit** : représente l'enfant droit ou `None` (si l'instance est une feuille).
  — **frequence** : représente la fréquence d'occurrence de ce caractère dans la chaîne initiale (si l'instance est une feuille) ou la somme des fréquences de tous ses descendants (si l'instance n'est pas une feuille).
- Méthodes :
  — **estFeuille(self)** : renvoie un booléen, `True` si l'instance est une feuille, sinon `False`.

Dans la suite de l'énonce, le processus de création de cet arbre de Huffman est expliqué par un exemple d'un texte composé seulement des caractères suivants : A, B, C, D et E.

La génération de codes de Huffman passe par les étapes suivantes :

Étape 1. Calcul des fréquences des symboles
Pour commencer, il faut calculer la fréquence de chaque symbole dans la séquence de données à compresser. Cela implique de compter le nombre d'occurrences de chaque symbole.
Question 1. Créer une fonction python **`occurrences(texte)`** qui calcule et retourne la fréquence de chaque caractère du `texte` passé en paramètre et renvoie le résultat sous forme de dictionnaire.
Exemple : `freqChar = occurrences(texte)` renvoie le dictionnaire `freqChar = {'A':12, 'B':15, 'C':5, 'D':13:, 'E':9}`

Étape 2. Construction de l'arbre de Huffman
En utilisant les fréquences des symboles calculées à l'étape précédente, un arbre binaire est construit. Dans cet arbre :

- Chaque *feuille* représente un symbole et sa fréquence.
- La fréquence de chacun des autres nœuds est la somme des fréquences de ses descendants.

Question 2. Créer le constructeur de la classe **`Arbre`**. Veuillez bien à respecter cette signature :
`def __init__(self, char="", frequence=0, nœudGauche=None, nœudDroit=None)`

Question 3. Créer une méthode **`estFeuille(self)`** permettant de retourner `True` si le nœud est sans descendants, `False` sinon.

Question 4. Créer une méthode **`__lt__(self, other)`** permettant de comparer si l'arbre `self` est inférieur à l'arbre `other`. Tel que, si A et B deux arbres, $A < B$ retourne `True` si la fréquence de A est inférieure à celle de B, ou s'ils ont la même fréquence et le code $ASCII$ de A est inférieur à celui de B. Sinon elle retourne `False`.

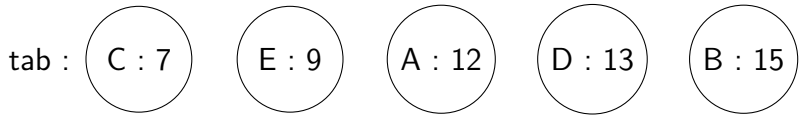
Étape 2.1. Initialisation
L'algorithme de création de cet arbre de Huffman commence par créer *les feuilles de l'arbre* pour chaque symbole du texte à coder, chacune marquée par sa fréquence, et les sauvegarder dans un tableau trié.

Question 5. Écrivez une fonction **`inserer_arbre(tab, A)`** qui insère un arbre **`A`** dans un tableau **`tab`** contenant des objets de type Arbre. Ce tableau est trié par ordre croissant en fonction de la valeur de la fréquence des nœuds racines de ces arbres.

Question 6. Écrivez une fonction `creer_feuilles(freqChar)` qui crée et retourne un tableau contenant des objets de type Arbre élémentaire pour chaque caractère du texte, à partir du dictionnaire des fréquences `freqChar` passé en paramètre. Le tableau retourné doit être trié par fréquence croissante, puis par ordre ASCII croissant en cas d'égalité de fréquence. Exemple :

```
tab = creer_feuilles({'A':12, 'B':15, 'C':7, 'D':13:, 'E':9})
```

Création d'un tableau trié `tab` d'arbres élémentaires (feuilles), un arbre élémentaire par symbole :



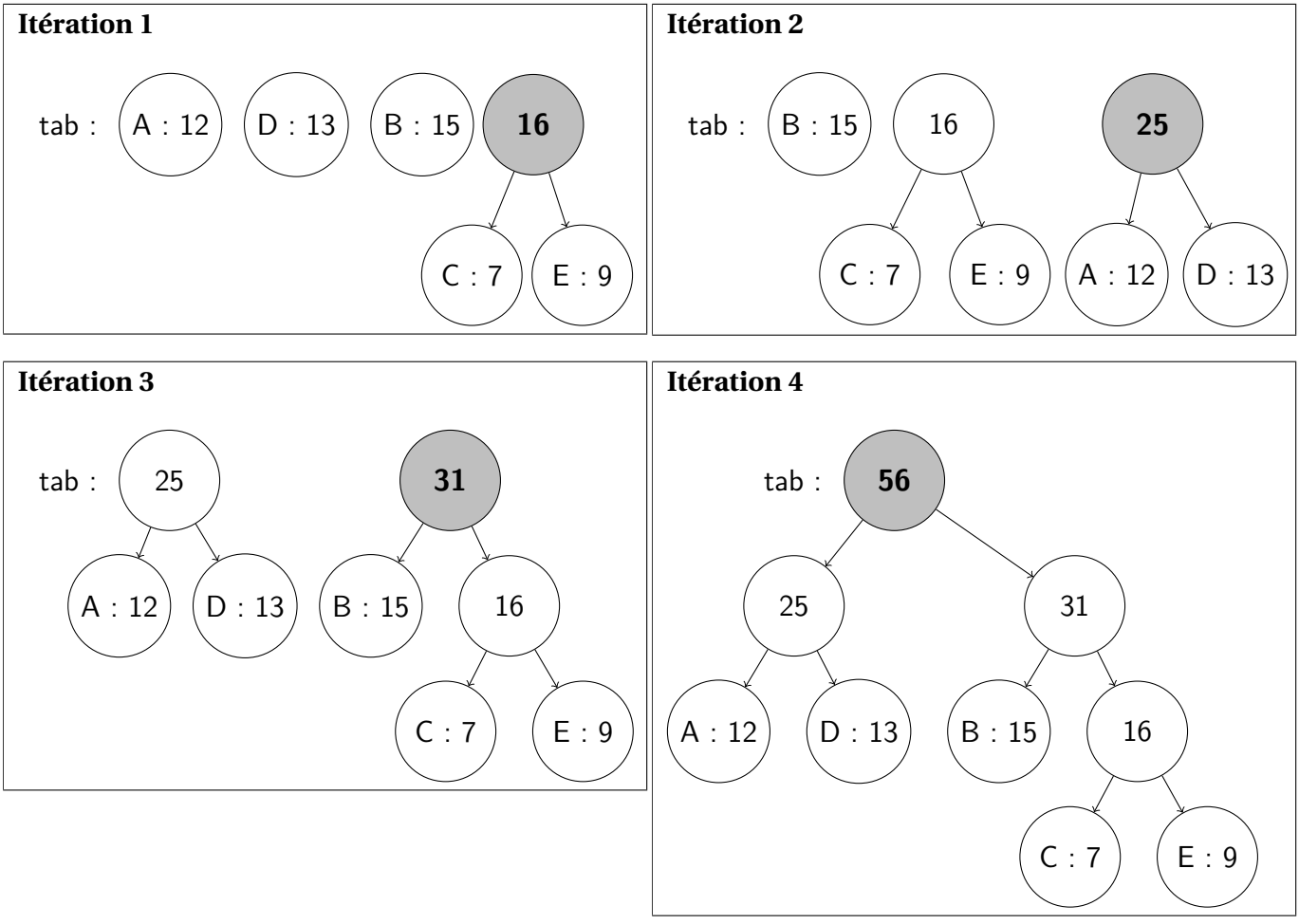
La feuille située à l'indice 0 dans le tableau `tab` représente le caractère $'C'$ et a une fréquence de 7.

Étape 2.2. Fusions des nœuds

Dans chacune des itérations suivantes, on procède comme suit :

— Retirer les deux premiers arbres du tableau (ceux ayant les fréquences les plus basses).
— Créer un nouvel arbre à partir de ces deux arbres retirés, dont la fréquence est égale à la somme des fréquences des deux arbres retirés.
— Insérer le nouvel arbre ainsi créé à la bonne position dans le tableau.

Les itérations se terminent lorsqu'il ne reste qu'un seul arbre dans le tableau.

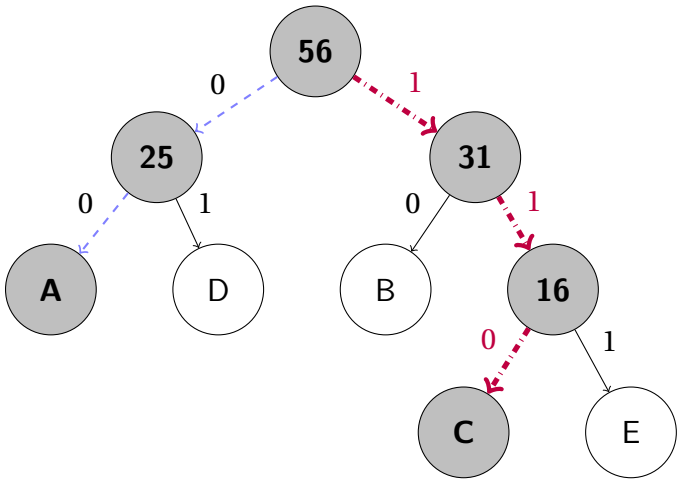


Question 7. Écrivez la fonction `créer_arbre(tab)` qui génère et retourne un arbre de Huffman à partir d'un tableau d'arbres élémentaires triés `tab`.

Etape Finale. Génération du dictionnaire de codage

La représentation binaire de chaque caractère est simplement le chemin depuis la racine jusqu'à la feuille correspondant au caractère, en utilisant "0" pour la branche de gauche et "1" pour la branche de droite.

Par exemple :
Le code binaire pour "A" est `00`.
Le code binaire pour "C" est `110`.



À la fin du processus, un **dictionnaire de codage** est généré. Ce dictionnaire associe chaque symbole à son code de Huffman correspondant, utilisé ensuite pendant la compression pour substituer chaque occurrence d'un symbole par son code de Huffman associé.

Question 8. Écrivez la fonction `codeBinaire(arbre, dico={}, code="")` qui permet de remplir le dictionnaire `dico` avec les caractères des feuilles de l'arbre de Huffman et leurs codes binaires correspondants.
Exemple : `codeBinaire(arbre)` remplit le dictionnaire `dico` avec les valeurs suivantes : `{'A':'00', 'C':'110', 'B':'10', 'D':'01', 'E':'111'}`

Compression de la séquence de données :
En utilisant le dictionnaire de codage généré, chaque symbole dans la séquence de données initiale est remplacé par son code de Huffman correspondant, ce qui donne une séquence de bits compressée.

Question 9. Implémentez la fonction `compresser(texte, dico)` qui retourne la séquence binaire du `texte` codé. La fonction devrait également afficher le taux de compression.
Exemple :

```
text="BADDABCADEABEBBADDBBDBEBBCEDBDBCBABBDBCEADEEDBBAACABCDEA" # len(text)==56
dico = {'A':'00', 'C':'110', 'B':'10', 'D':'01', 'E':'111'}
code = compresser(text)
taux de compression = 71.88 %
print(code)
"0b1000010100101100001111001011110100001011010011011110101101110 1
1001101101000101001101101110001111111011010000011000101100111100"
```

Décompression de la séquence de bits :
Question 10. Écrivez la fonction `decompresser(code, dico)` qui restaure le texte initial à partir du `code` binaire compressé, en utilisant le dictionnaire de codage `dico` utilisé pour la compression, où chaque séquence de bits correspond à un caractère spécifique.

Bon travail**.**