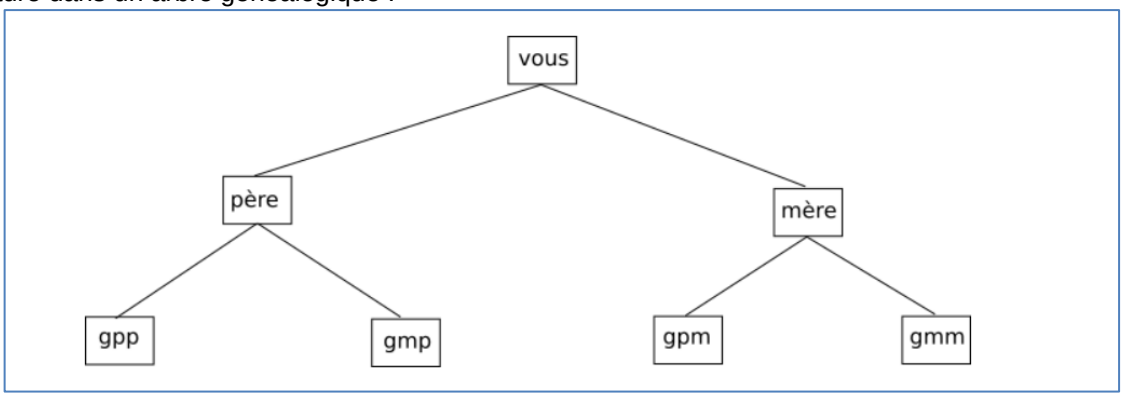
1. Introduction

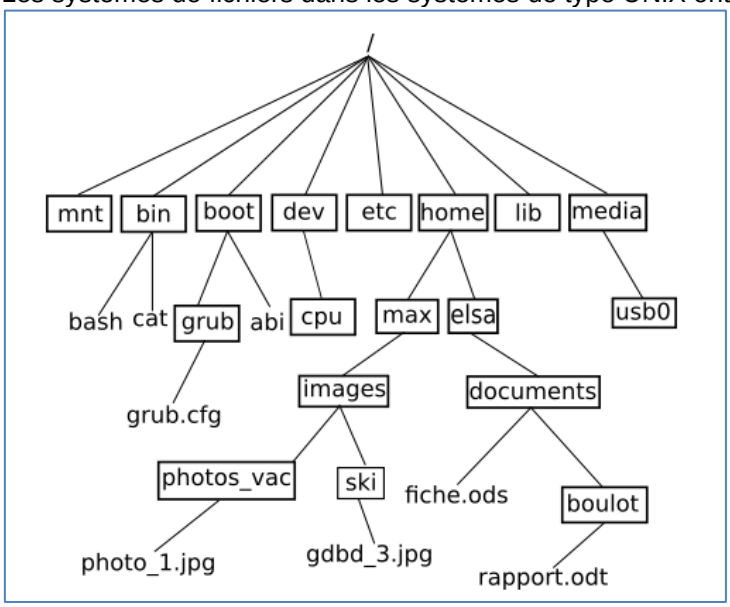
Les arbres sont très utilisés en informatique, d'une part parce que les informations sont souvent hiérarchisées, et peuvent être représentées naturellement sous une forme arborescente, et d'autre part, parce que les structures de données arborescentes permettent de stocker des données volumineuses de façon que leur accès soit efficace.

Exemples :

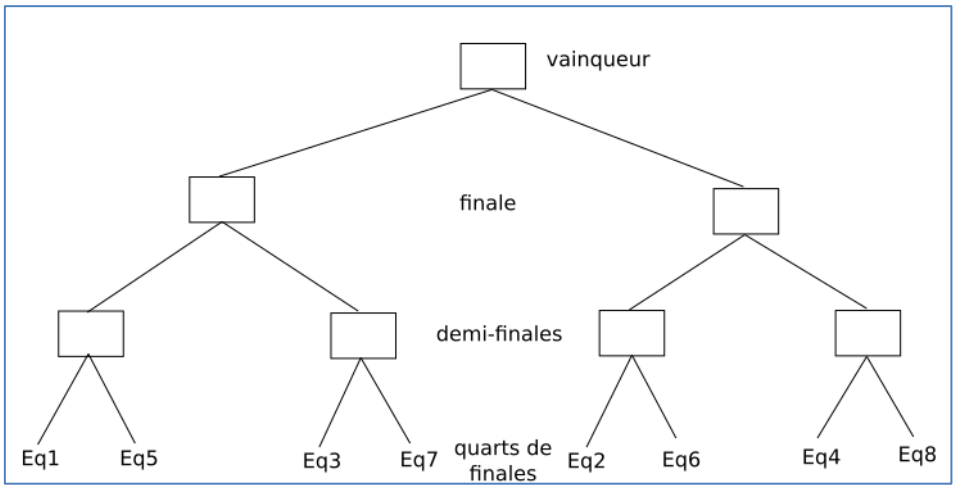
- Nous avons ci-dessous ce que l'on appelle une structure en arbre. On peut aussi retrouver cette même structure dans un arbre généalogique :



- Les systèmes de fichiers dans les systèmes de type UNIX ont aussi une structure en arbre.



- Tournois



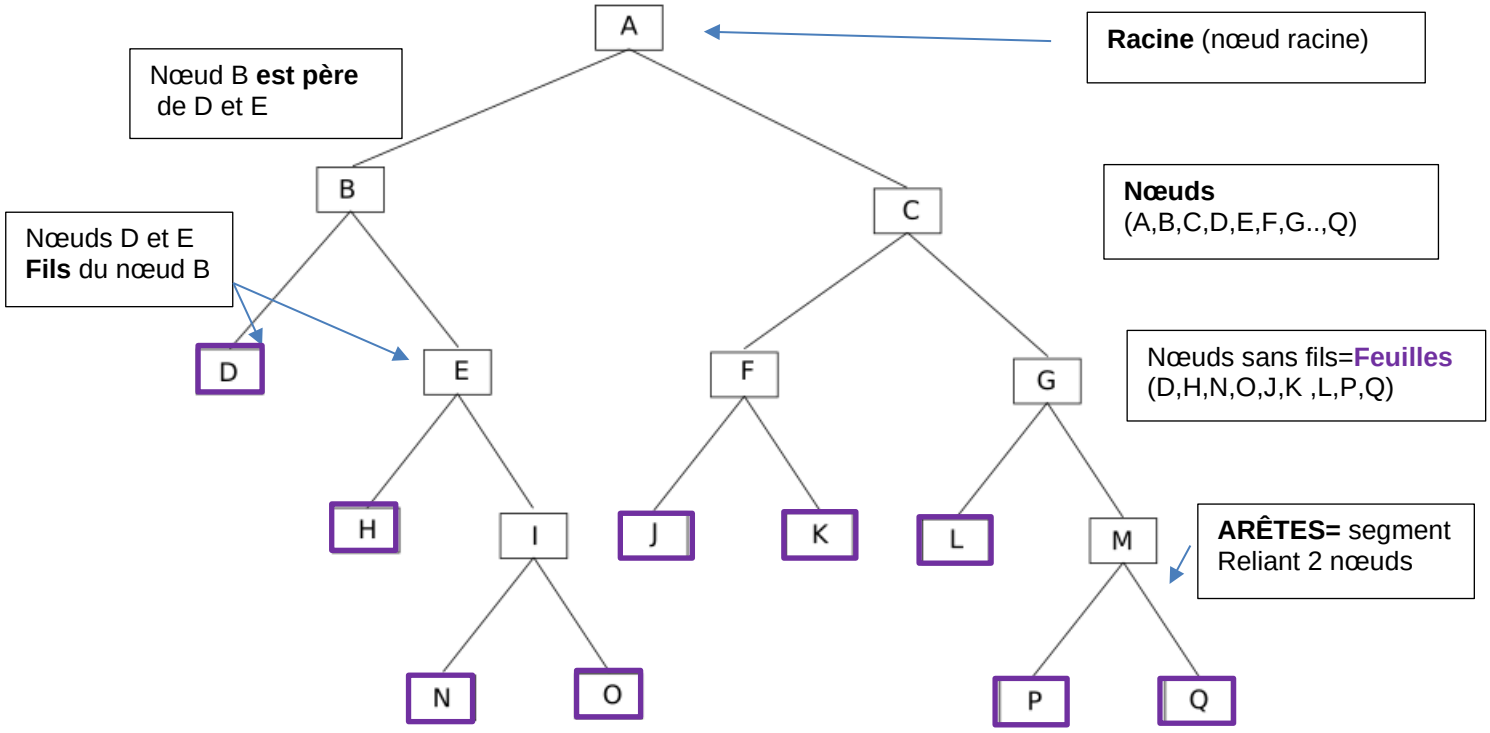
Les arbres binaires sont des cas particuliers d'arbre : l'arbre du tournoi et l'arbre "père, mère..." sont des arbres binaires, **en revanche**, l'arbre représentant la structure du système de fichier n'est pas un arbre

binaire.

Dans un arbre binaire, on a ***au maximum 2 branches*** qui partent d'un élément (pour le système de fichiers, on a 7 branches qui partent de la racine : ce n'est donc pas un arbre binaire). Dans la suite nous **allons uniquement travailler sur les arbres binaires**.

2. Arbre binaire

Vocabulaire :



- Un arbre binaire est défini de la façon suivante : ou bien il est vide, ou bien il est constitué d'un nœud racine qui possède deux fils, un fils gauche et un fils droit, qui sont tous les **deux des arbres binaires**. **Il s'agit donc d'une définition récursive.**

- **On appelle profondeur d'un nœud ou d'une feuille** dans un arbre binaire le nombre de nœuds du chemin qui va de la racine à ce nœud.

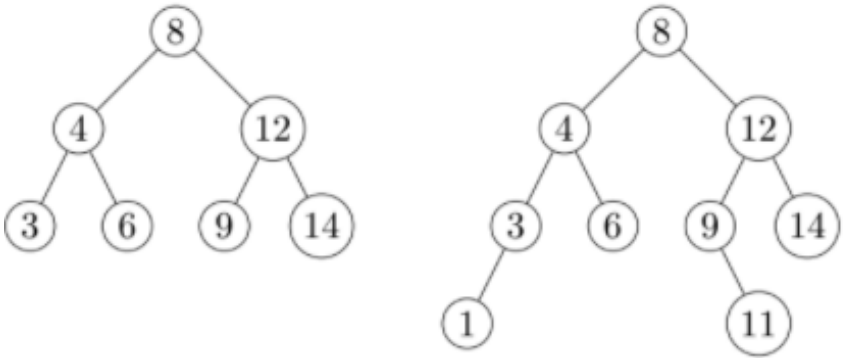
- **La racine d'un arbre** est à une **profondeur 1**, et la profondeur d'un nœud est égale à la profondeur de son prédécesseur plus 1. Si un nœud est à une profondeur p, tous ses successeurs sont à une profondeur p+1. Exemples : profondeur de B = 2 ; profondeur de I = 4 ; profondeur de P = 5

ATTENTION : on trouve aussi dans certains livres la profondeur de la racine égale à 0 (on trouve alors : profondeur de B = 1 ; profondeur de I = 3 ; profondeur de P = 4). Les 2 définitions sont valables, il faut juste préciser si vous considérez que la profondeur de la racine est de 1 ou de 0.

- On appelle **hauteur** d'un arbre la profondeur maximale des nœuds de l'arbre.
  Exemple : la profondeur de P = 5, c'est un des nœuds les plus profond, donc la hauteur de l'arbre est de 5.

ATTENTION : comme on trouve 2 définitions pour la profondeur, on peut trouver 2 résultats différents pour la hauteur : si on considère la profondeur de la racine égale à 1, on aura bien une hauteur de 5, mais si l'on considère que la profondeur de la racine est de 0, on aura alors une hauteur de 4.

Considérons les deux arbres de la figure suivante.



Le premier a pour racine un nœud contenant l'entier 8, la racine de son fils gauche contient l'entier 4, la racine de son fils droit contient l'entier 12.
Remarque : *Deux nœuds différents peuvent contenir la même valeur.*

- Le nombre de nœuds d'un arbre binaire s'appelle sa **taille**. Ainsi la taille de l'arbre de gauche est égale à 7, celle de l'arbre de droite est égale à 9.

3. Représentation d'un arbre sous python

Nous représenterons les arbres en Python comme suit :

- L'arbre vide est représenté par None
- Si t est un arbre non vide, de racine x, de fils gauche u et de fils droit v nous représenterons t par le triplet (x, u, v).

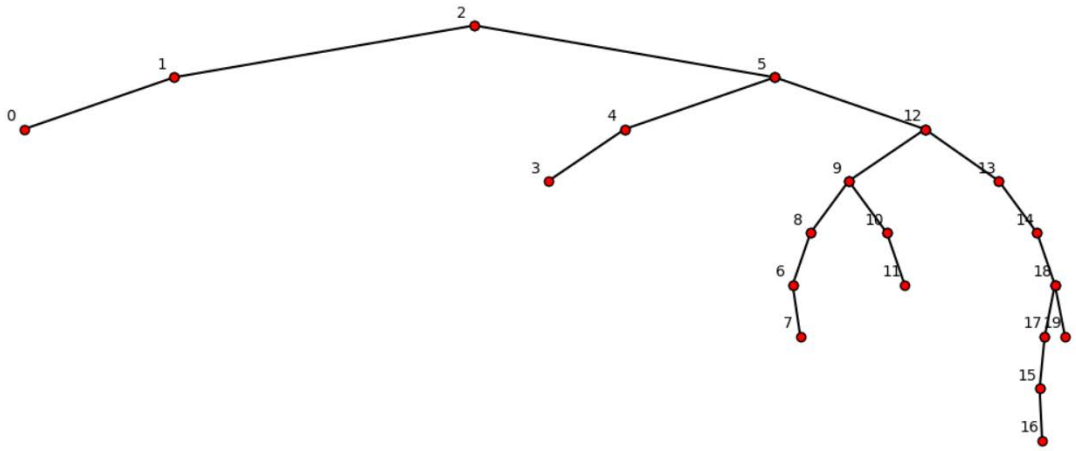
a) Importer les librairies suivantes :

```
import matplotlib.pyplot as plt
import random
plt.rcParams['figure.figsize'] = (20, 6)
```

L'arbre None sera appelé l'arbre vide.
Les fonctions ci-dessous explicitent ces définitions, déclarez ces différentes fonctions :

```
def vide(): return None
def racine(t):return t[0]
def fg(t):return t[1]
def fd(t): return t[2]
def fils(t):return(gauche(t),droit(t))
def est_vide(t):return t==None
def arbre(x,u,v):return (x,u,v)
```

Jusqu'à ce que nous puissions faire mieux, voici un arbre que nous utiliserons pour tester nos fonctions.



Déclarer l'exemple ci-dessous qui correspond au graphe représenté au-dessus :

```
exemple=(2,(1,(0,None,None),None),(5,(4,(3,None,None),None),
(12,(9,(8,(6,None,(7,None,None)),None),
(10,None,(11,None,None))),(13,None,(14,None,(18,(17,(15,None,(16,None,None)),
None),(19,None,None)))))))
```

Vous sauvegarderez sous arbres.py tout au long de la séquence.

b) Hauteur, nombre de Nœuds, feuilles d'un arbre

Définition (Noeud) : Soit *t* un arbre. Si *t* est vide, alors *t* n'a pas de nœud. Sinon, *t* = (*x*, *u*, *v*) où *u* et v sont des arbres. Un nœud de *t* est alors *x* ou un nœud de *u* ou un nœud de *v*.

À titre d'exemple, écrivons une fonction récursive qui calcule le nombre de nœuds d'un arbre.

```
def nombre_noeuds(t):
    if est_vide(t): return 0
    else:
        _,u,v=racine(t),fg(t),fd(t)
        return nombre_noeuds(u)+nombre_noeuds(v)+1
```

Testons cette fonction sur notre arbre

```
print(nombre_noeuds(exemple)
```

Donner la valeur du nombre de nœuds affiché

Définition (Feuille) : soit *t* un arbre. Si *t* est vide alors *t* n'a pas de feuille. Sinon, *t* = (*x*, *t*1, *t*2) où *t*1 et *t*2 sont des arbres. Si *t*1 et *t*2 sont vides alors *x* est une feuille de *t*. Sinon, les feuilles de *t* sont les feuilles de *t*1 ou les feuilles de *t*2.

Ecrire sous python la fonction qui donne **une liste des feuilles de l'arbre :**

```
def liste_feuilles(t):
    if est_vide(t): return []
    else:

        x, u, v = racine(t),fg(t),fd(t)
        if est_vide(u) and est_vide(v): return [x]
        else: return liste_feuilles(u) + liste_feuilles(v)
```

Testons cette fonction sur notre arbre

```
print(liste_feuilles(exemple))
```

Donner la valeur de la liste.

Ecrire sous python la fonction qui donne **le nombre de feuilles de l'arbre** :

```
def nombre_feuilles(t):
    if est_vide(t):return 0
    else:
        _,u,v=racine(t),fg(t),fd(t)
        if est_vide(u) and est_vide(v):return 1
        else: return nombre_feuilles(u)+nombre_feuilles(v)
```

Testons cette fonction sur notre arbre

```
print(nombre_feuilles(exemple)
```

Donner la valeur du nombre de feuilles affichés.

Définition (Hauteur) : Soit *t* un arbre. Si *t* est vide, sa hauteur est 0. Sinon, *t* = (*x*, *t*1, *t*2) où *t*1 et *t*2 sont des arbres. La hauteur de *t* est alors 1 + le max de la hauteur de *t*1 et de la hauteur de *t*2. La hauteur de *t* est la longueur du plus long chemin de sa racine à l'une de ses feuilles.

Ecrire sous python la fonction qui donne **la hauteur de l'arbre** :

```
def hauteur(t):
    if est_vide(t): return 1
    else:
        u,v=fg(t),fd(t)
        return 1 +max(hauteur(u),hauteur(v))
```

Testons cette fonction sur notre arbre

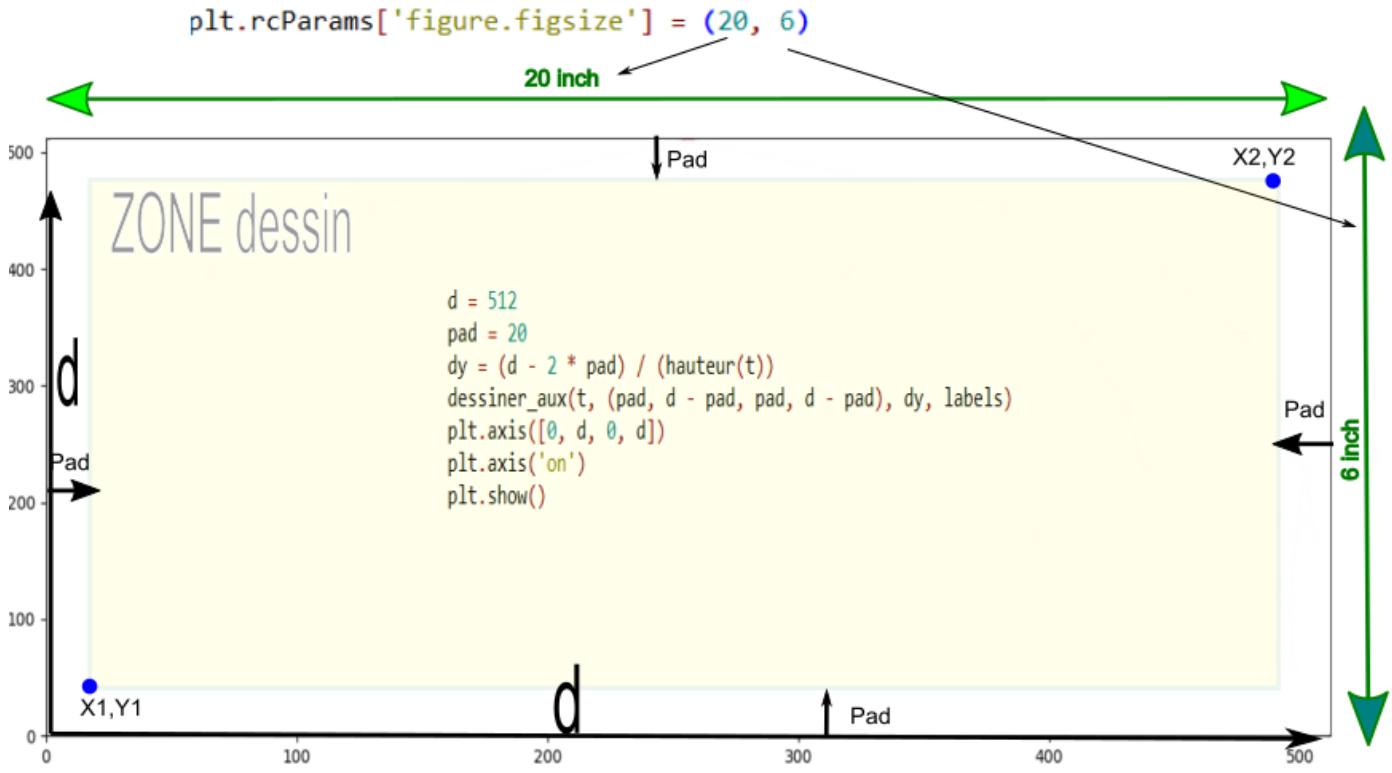
```
print("hauteur?",hauteur(exemple))
```

Donner la valeur de la hauteur affichée.

c) Dessiner un arbre :

Pour dessiner cet arbre il faut :

- On dessine son fils gauche
- On dessine son fils droit
- On trace un trait de la racine au fils gauche
- On trace un trait de la racine au fils droit
- Si l'arbre est vide on ne dessine rien



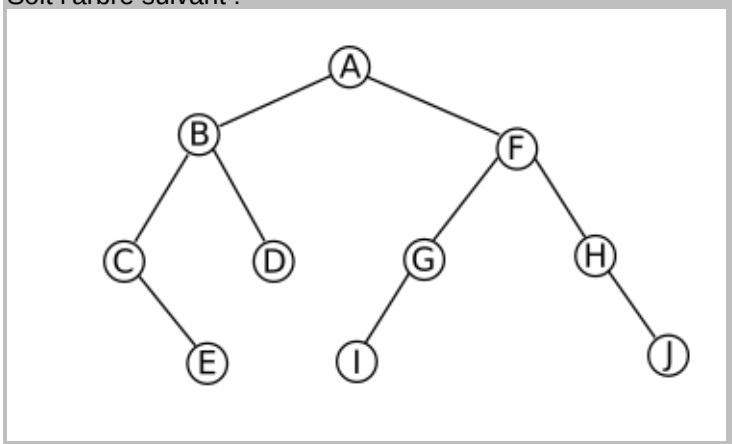
Voici la fonction de tracé. Elle prend un arbre *t* en paramètre. Elle initialise (un carré où tracer l'arbre. Elle calcule la distance idéale entre le tracé de deux niveaux de l'arbre. Cette distance dépend bien entendu de la hauteur de l'arbre *t*. Enfin elle appelle **dessiner_aux(t,rect,dy,labels).**

```
def dessiner(t, labels=True):
    d = 512
    pad = 20
    dy = (d - 2 * pad) / (hauteur(t))
    dessiner_aux(t, (pad, d - pad, pad, d - pad), dy, labels)
    plt.axis([0, d, 0, d])
    plt.axis('off')
    plt.show()
```

```
def dessiner_aux(t, rect, dy, labels):
    if est_vide(t): return
    x1,x2,y1,y2=rect
    xm=(x1+x2)//2
    x,t1,t2=t

    dessiner_aux(t1,(x1,xm,y1,y2-dy),dy,labels)
    dessiner_aux(t2,(xm,x2,y1,y2-dy),dy,labels)
    if labels:
        plt.text(xm-5,y2+5,str(x),fontsize=10,horizontalalignment='center',va='bottom')

    if not est_vide(t1):
        a,b=((xm,(x1+xm)//2),(y2,y2-dy))
        plt.plot(a,b,'k',marker='o',markerfacecolor='r')

    if not est_vide(t2):
        c,d=((xm,(x2+xm)//2),(y2,y2-dy))
        plt.plot(c,d,'k',marker='o',markerfacecolor='r')
```

Coder et tester cette fonction sur notre arbre : dessiner (exemple)
Capturer l'image de l'affichage de cet arbre.

Soit l'arbre suivant :



Proposer une déclaration sous forme de tuple de la même structure que le tuple exemple
Vous l'appellerez exemple2
exemple2=('A',……
……

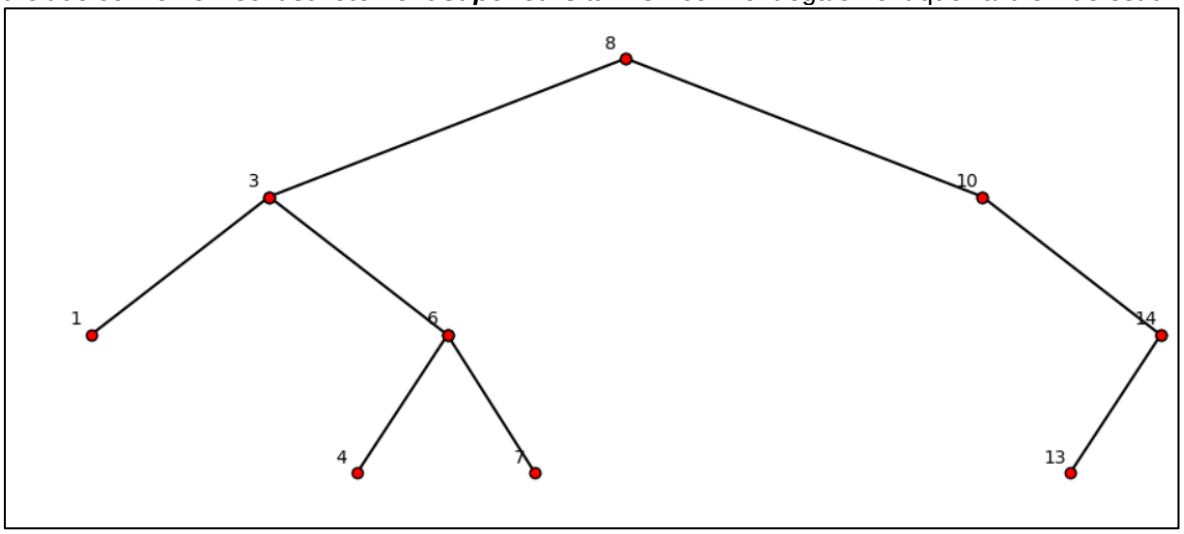
Donner :

- le nombre de nœuds de exemple2
- le nombre de feuilles de exemple2
- la hauteur de l'arbre exemple2
- le résultat d'affichage de exemple2

en Utilisant les fonctions créées précédemment pour répondre aux différentes questions.

4. Arbres binaires de recherche

Nous allons-nous intéresser à des arbres binaires dans lesquels il est facile de retrouver un nœud. On les appelle **des arbres binaires de recherche** (ABR en abrégé). Ce sont les arbres vérifiant, pour chaque nœud *x* de l'arbre, que les nœuds du fils gauche de *x* sont ***inférieurs ou égaux*** à *x*, et les nœuds du fils droit de ce même *x* sont ***strictement supérieurs à x***. On convient également que l'arbre vide est un ABR.



Créer cet arbre exemple3 sous python et vérifier en utilisant la fonction dessiner (exemple3) que l'affichage est correct
Capturer le résultat.

a) Recherche un élément dans un arbre

Il est facile de rechercher un objet dans un ABR. Si l'élément est plus petit que la racine, on recherche dans le fils gauche sinon on recherche dans le fils droit on sort de la fonction soit par **True** ou **False** en fonction du résultat de la recherche.

```
def rechercher(x, t):
    if est_vide(t): return False
    else:
        y, u, v = racine(t), fg(t), fd(t)
        if x < y: return rechercher(x, u)
        elif x > y: return rechercher(x, v)
        else: return True
```

Coder cette fonction
Tester votre fonction, répondez alors aux questions ci-dessous :
Donner le résultat des recherches suivantes :
print(rechercher(7,exemple3)) = ……
print (rechercher(20,exemple3)) =……

b) Minimum, Maximum , d'un arbre

Le maximum d'un ABR est facile à trouver. On part de la racine et on va à droite, à droite, . . . Bref c'est à fond à droite.

```
def maximum(t):
    if est_vide(t):raise Exception('Arbrevide')
    else:
        x,_,v=racine(t),fg(t),fd(t)
        if est_vide(v):return x
        else:return maximum(v)
```

Pour le minimum, on va à fond à gauche :

```
def minimum(t):
    if est_vide(t):raise Exception('Arbrevide')
    else:
        x,u,_=racine(t),fg(t),fd(t)
        if est_vide(u):return x
        else:return minimum(u)
```

Coder ces fonctions
Donner le résultat des fonctions maximum(t) et minimum(t) appliqués à l'arbre exemple3

```
print(maximum(exemple3))
print(minimum(exemple3))
```

c) Modification d'un arbre

Un arbre binaire de recherche peut être vu comme une structure qui permet de stocker un ensemble de données. Où cela ? Dans les nœuds, bien entendu. Nous venons de voir comment rechercher une donnée dans cet ensemble efficacement (si tout va bien). Cela dit, les ensembles de données ne sont intéressants que s'ils peuvent évoluer au cours du temps. On veut pouvoir ajouter et supprimer des éléments dans l'ensemble. Il doit être une structure de données **dynamique**. Les trois opérations fondamentales sur une structure d'ensemble dynamique sont :

- RECHERCHER (déjà vu précédemment)
- INSERER
- SUPPRIMER

i. INSERER

Insérer un Nœuds n'est pas plus difficile que rechercher :

```
def inserer(x, t):
    if est_vide(t): return (x, vide(), vide())
    else:
        y, u, v =racine(t),fg(t),fd(t)
        #if x==y: return (y,u,v)    partie grisée à implémenter si on veut éviter les doublons
        # if x<y: return (y, inserer(x, u),v)
        if x <= y: return (y, inserer(x, u),v)# à supprimer si on veut eviter les doublons
        else: return (y,u, inserer(x, v))
```

Coder cette fonction
Tester la fonction d'insertion qui permet d'insérer 1 dans l'exemple3. (Il y aura alors un doublon). Capture le résultat obtenu

ABR aléatoire

Nous voici prêts à créer de façon automatique des arbres un peu plus gros
Pour créer un ABR "aléatoire", on crée une liste aléatoire et on insère ses éléments dans un ABR initialement vide.

```
def liste_aleatoire(n):
    s=list(range(n))
    random.shuffle(s)
    return s
```

Donner le résultat de la liste obtenue par

```
print (liste_aleatoire(15))
```

Créons maintenant un arbre aléatoire à partir de cette liste

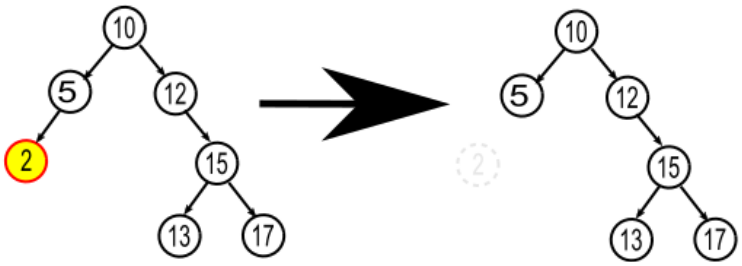
```
def abr_aleatoire(n):
    s=liste_aleatoire(n)
    t=None
    for x in s:
        t=inserer (x,t)
    return t
```

Donner le résultat de l'affichage d'un ABR aléatoire de 50 nœuds.

ii. SUPPRIMER

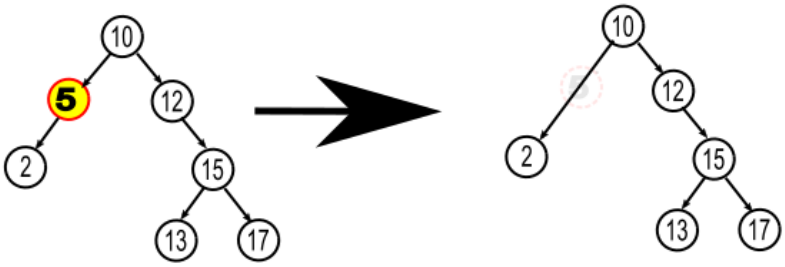
On désire supprimer x dans l'ABR t.Que peut-il arriver ?

Si x n'a pas de descendants (c'est une feuille) on supprime dans le fils gauche ou le fils droit de t.

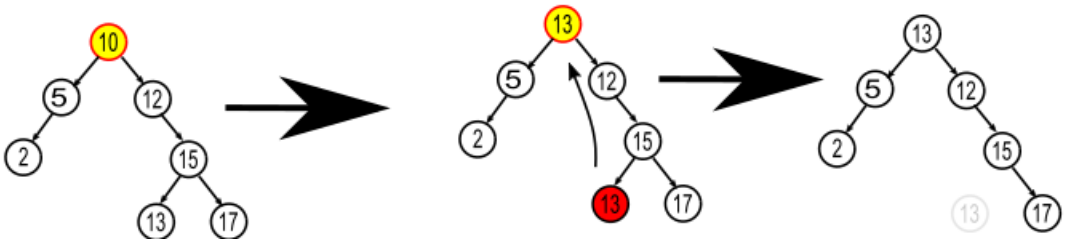


Sinon :

Si x n'a pas de fil droit, c'est facile, renvoyer le fils gauche de t.



Si x a aussi un fils droit, soit le minimum de ce fils droit. Remplacer x par m son **successeur=*le nœud le plus à gauche du sous arbre droit*** qui est toujours le **minimum de ses descendants droits** et supprimer récursivement m du fils droit de x.

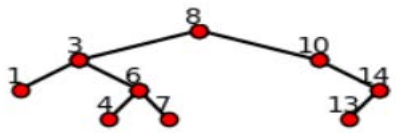
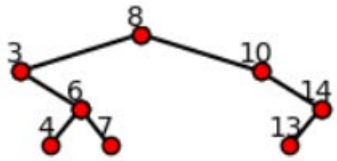
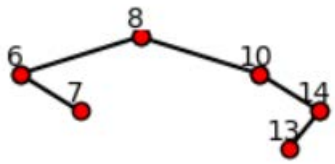
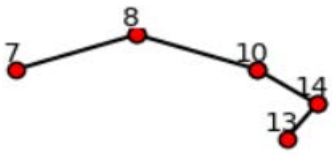
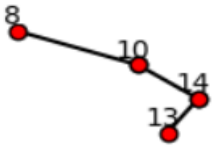
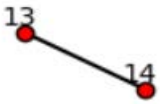



```
def supprimer(x, t):
    if est_vide(t): return None
    else:
        y, u, v = racine(t),fg(t),fd(t)
        if x < y: return (y, supprimer(x, u), v)# si x est inférieure à la valeur du noeud, rechercher dans le sous-arbre gauche
        elif x > y: return (y, u, supprimer(x, v))# si x est supérieure à la valeur du noeud, rechercher dans le sous-arbre droit
        elif est_vide(v):return u                   # On a trouvé le noeud (x==y) s'il n'a pas de fils droit on retourne le noeud fils gauche
        else:                           #  x a un fils droit
            m = minimum(v)              # rechercher successeur (minimum du fils droit)
            return (m, u, supprimer(m, v))  # on remplace x par m et on supprime récursivement m
                                            #du fils droit de x
```

Coder cette fonction

Tester cette fonction sur l'ABR exemple3. Et supprimer le nœuds 3 puis afficher l'arbre :
Dessiner(supprimer(3,exemple3))
Capturer le résultat

A titre d'exemple on montre ci-dessous l'évolution de la suppression des différents nœuds de l'arbre exemple3

 Abre de départ	 Suppression 1	 Suppression 3
 Suppression 4	 Suppression 6	 Suppression 7
 Suppression 8	 Suppression 10	14

d) Parcourir un arbre binaire

Parcourir un arbre, c'est explorer, visiter, tous ses nœuds. Pour quoi faire ? Tout un tas de choses. En réalité, un grand nombre des fonctions que nous avons écrites exploraient des arbres : hauteur, nombre de nœuds, affichage, etc. Il existe plusieurs façons de parcourir un arbre, dont, entre autres :

PARCOURS EN LARGEUR

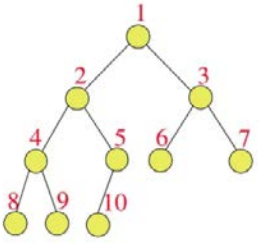
Le parcours niveau par niveau. Racine, puis fils de la racine, puis petits-fils, etc.

PARCOURS EN PROFONDEUR

Le parcours préfixe : on visite la racine, puis on parcourt le fils gauche, puis on parcourt le fils droit.

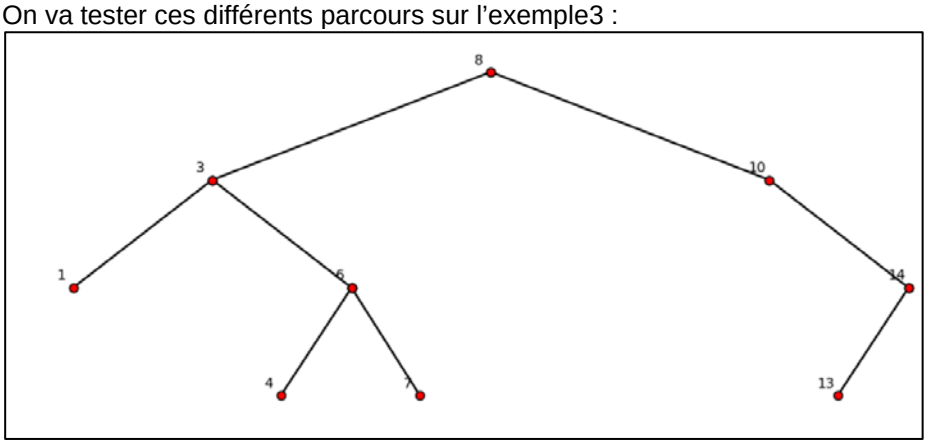
Le parcours infixe : on parcourt le fils gauche, puis on visite la racine, puis on parcourt le fils droit.

Le parcours suffixe : (appelé aussi postfixe) on parcourt le fils gauche, puis on parcourt le fils droit, puis on visite la racine.



Préfixe : 1, 2, 4, 8, 9, 5, 10, 3, 6, 7 (VGD)
Infixe : 8, 4, 9, 2, 10, 5, 1, 6, 3, 7 (GVD)
Suffixe : 8, 9, 4, 10, 5, 2, 6, 7, 3, 1

On va tester ces différents parcours sur l'exemple3 :



i. Parcours préfixe :

```
Algorithme prefixe(Noeud)
    1. Visitez la racine.
    2. Parcourez le sous-arbre gauche, c'est-à-dire, appelez prefixe(sous-arbre gauche)
    3. Parcourez le sous-arbre droit, c'est-à-dire appelez prefixe(sous-arbre droit)
```

```
def parcours_prefixe(t):
    if est_vide(t):return []
    else:
        x,u,v=racine(t),fg(t),fd(t)
        return [x]+parcours_prefixe(u)+parcours_prefixe(v)
```

Coder cette fonction
Tester cette fonction sur l'ABR exemple3.

```
print(parcours_prefixe(exemple3))
```

Capturer le résultat par le parcours préfixe.

Le parcours préfixe est utilisé pour créer une copie de l'arbre.

ii. Parcours infixe :

```
Algorithme infixe (Noeud)
    1. Parcourez le sous-arbre gauche, c'est-à-dire, appelez infixe(sous-arbre gauche)
    2. Visitez la racine.
    3. Parcourez le sous-arbre droit, c'est-à-dire appelez infixe(sous-arbre droit)
```

```
def parcours_infixe(t):
    if est_vide(t):return []
    else:
        x,u,v=racine(t),fg(t),fd(t)
        return parcours_infixe(u)+[x]+parcours_infixe(v)
```

Coder cette fonction
Tester cette fonction sur l'ABR exemple3.

```
print(parcours_infixe(exemple3))
```

Capturer le résultat par le parcours infixe.
Que remarquez-vous sur l'ordre des nœuds dans ce cas ?

iii. Parcours suffixe :

```
Algorithme postfixe(Noeud)
    1. Parcourez le sous-arbre gauche, c'est-à-dire, appelez postfixe(sous-arbre gauche)
    2. Parcourez le sous-arbre droit, c'est-à-dire appelez postfixe(sous-arbre droit)
    3. Visitez la racine.
```

```
def parcours_suffixe(t):
    if est_vide(t):return []
    else:
        x,u,v=racine(t),fg(t),fd(t)
        return parcours_suffixe(u)+parcours_suffixe(v)+[x]
```

Coder cette fonction
Tester cette fonction sur l'ABR exemple3.

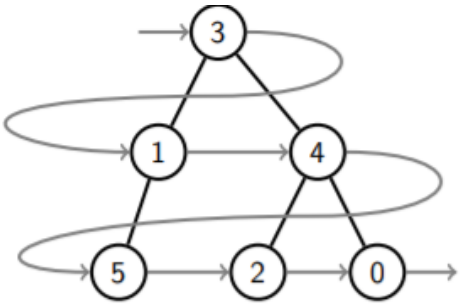
```
print(parcours_suffixe(exemple3))
```

Capturer le résultat par le parcours suffixe.

Le parcours suffixe est utilisé pour supprimer l'arbre

iv. Parcours en largeur :

Nous allons aborder un type de parcours un peu plus compliqué, c'est le parcours en largeur. La récursivité ne peut s'appliquer.



Une méthode pour réaliser un parcours en largeur consiste à utiliser une structure de données de type file d'attente. Pour ceux qui ne connaissent pas encore ce type de structure de données, il s'agit tout simplement d'une structure de données qui est obéit à la règle suivante : premier entrée, premier sorti.

La file sera gérée par la librairie queue

```
from queue import *
def parcours_largeur(t):
    f=Queue(nombre_noeuds(t)) #declaration d'une file de la taille du nombre de noeuds
    f.put(t) #on met l'arbre entier dans la file (1 élément)

    while not f.empty():
        a=f.get() # on retire le premier élément de la file
        x,u,v=racine(a),fg(a),fd(a)
        print('x',x,'u',u,'v',v)
        if u is not None:   # la partie gauche de l'arbre est mis en premier
            f.put(u)
        if v is not None:    #la partie droite est mis en second
            f.put(v)

    return
```

Coder cette fonction
Tester cette fonction sur l'ABR exemple3. parcours_largeur(exemple3)

v. TRIER

Le parcours infixe nous donnait une liste triée des nœuds de l'arbre

```
print(parcours_infixe(exemple3))
```

Voici la réciproque de cette fonction qui permet à partir d'une liste triée de créer un arbre.

```
def liste_vers_arbre(s):
    t= None
    for x in s:
        t=inserer(x,t)
    return t
```

Coder cette fonction
Tester cette fonction pour créer, afficher un arbre à partir d'une liste aléatoire de 10 éléments
dessiner (liste_vers_arbre(liste_aleatoire(10)))
Capturer le résultat

Un algorithme de tri efficace pourrait alors s'écrire

```
def trier(s):
    return parcours_infixe(liste_vers_arbre(s))
```

Eh oui, on prend la liste *s*, on fabrique un ABR dont les nœuds sont ses éléments, on renvoie la liste des nœuds de l'ABR avec un parcours infixe. On a trié la liste *s*.
Coder cette fonction

Sur une liste triée.

```
s=liste_aleatoire(20)
print(s)
print(trier(s))
```

Dessiner (liste_vers_arbre(range(10)))
Capturer le résultat

L'algorithme de trie est relativement efficace si l'arbre n'est pas déséquilibré cas d'une liste déjà trié dans ce cas la complexité est en $O(n^2)$ sinon sa complexité moyenne est en **O(nlog(n))**.

5. Utilisation des arbres pour la compression de données :

a) Introduction

Une étape essentielle dans le stockage ou la transmission de l'information est le **codage** des mots. Pour être stocké ou transmis***, un mot doit être codé***. Et ce que nous obtenons après codage doit pouvoir être décodé (c'est préférable). Pour ceux d'entre vous qui se disent qu'ils ne stockent jamais de mots, dites-vous qu'un fichier n'est qu'un long mot dont les lettres sont, par exemple, les octets du fichier.

Le code ASCII

ASCII signifie "American Standard Code for Information Interchange". Le code ASCII a été développé dans les années 1960. C'est une norme de codage des caractères. Le code ASCII "pur" code 128 caractères par des mots de 7 bits sur l'alphabet. Par exemple, la lettre a est codée 1100001, la lettre b est codée 01100010 , etc. En interprétant ces mots de comme des entiers en base 2, on retient plutôt que le code ASCII de a est 97, celui de b est 98, etc.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

La fonction Python **ord()** permet d'obtenir l'entier numérique d'un caractère ASCII.

Coder les instructions suivantes :

```
for c in 'texte':
    print(c,ord(c))
```

Sauver sous compression.py, tester et capturer le résultat.

Sa réciproque est la fonction **chr()**

Coder l'instruction suivante :

```
print([(k, chr(k)) for k in range(256)])
```

Sauver sous compression.py, tester et capturer le résultat.

b) Coder mot ASCII en binaire

Ecrivons tout d'abord une fonction qui permet de coder un entier n en binaire sur 8 bits.

Coder la fonction suivante :

```
def entier_vers_binaire(n):
    s = ''                  #chaine vide
    for k in range(8):
        s = str(n % 2) + s #le nombre decimal soit pair ou impair
                           #est converti en chaine '0' ou '1' à chaque ittération
        n = n // 2         #division entiere de n par 2
    return s
```

Donner le résultat de la fonction pour n=100.

```
print(entier_vers_binaire(100))
```

Sauver sous compression.py, tester et capturer le résultat.

On peut maintenant donner sous forme binaire un texte complet

Coder la fonction suivante :

```
def coder_ascii(u):
    s = ''
    for a in u:
      s = s + entier_vers_binaire(ord(a))
    return s
```

Sauver sous compression.py.

Tester cette fonction sur le mot 'texte' et donner la longueur du texte en nombre de bits et capturer le résultat.

```
v='texte'
print(coder_ascii(v),len(coder_ascii(v)))
```

Représentation d'un code sous Python :

On peut à partir d'un dictionnaire coder une expression : code = {'a': '00', 'b': '01', 'c': '10', 'd' : '11'}

Coder la fonction suivante :

```
def coder (s,code):
    s1=''
    for a in s:
        s1=s1+code[a]
    return s1
```

Sauver sous compression.py.

Testons cette fonction sur une chaine contenant des caractères appartenant au dictionnaire :

exemple : `print(coder ('abcd',code))`

Capturer le résultat.

Pour coder un texte ASCII plus complet en binaire à partir d'un dictionnaire il nous faut d'abord le créer la fonction suivante :

```
def sous_ascii():
    c={}
    s='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz 0123456789,;:.'
    for a in s:
        c[a]=entier_vers_binaire(ord(a))
    return c
```

Sauver sous compression.py.

Tester la fonction et donner le résultat du dictionnaire obtenu en affichant le résultat de l'appel de celle-ci.

c) Décoder du binaire en Ascii

i. Comment décode-t-on ?

Essayons de fabriquer un dictionnaire inverse :

Coder la fonction suivante :

```
def inverser(code):
    d={}
    for a in code:
        d[code[a]]=a
    return d
```

Sauver sous compression.py.

Testons cette fonction sur notre dictionnaire ascii : `print(inverser(sous_ascii()))` et capturer le résultat

Décoder de l'ASCII c'est facile puisque les différents codes binaires tiennent toujours sur le même nombre de bits ici 8. Il suffit donc pour décoder de regrouper les bits par paquet de 8 et à partir du dictionnaire inversé ASCII de retrouver la lettre correspondante.

Coder la fonction suivante :

```
def decoder_ascii(s):
    s1='' #declaration d'un chaine vide pour le décodage
    for k in range (len(s)//8):  #on avance modulo 8
        a=s[8*k: 8*(k+1)] # on isole chaque paquet de 8bits
        s1=s1 +inv_mini_ascii[a] #on ajoute a la chaine le caractére correspondant au code 8 bits
    return s1
```

Sauver sous compression.py.

Testons cette fonction :

```
inv_mini_ascii = inverser(sous_ascii()) #stocker le dictionnaire inversé dans une variable
v=coder_ascii('Bonne chance')
print(v)
print(decoder_ascii(v))
```

Capturer le résultat.

On a bien compris maintenant que coder et décoder sur un format fixe (8bit) est relativement aisé. Ce n'est sûrement pas avec cette méthode que l'on pourrait réduire sans perte la taille d'un texte écrit en ASCII.

ii. Code préfixe :

Un code préfixe (ou **code instantané**) est un code ayant la particularité de ne posséder aucun mot du code ayant pour préfixe un autre mot du code. Autrement dit, aucun mot du code (ou symbole) d'un code préfixe ne peut se prolonger pour donner un autre mot du code (ou symbole).

Ce sont des codes non ambigus. Les codes à taille fixe sont tous des codes préfixes

Soit un dictionnaire utilisant le code ci-dessous : code={'a':1,'b':'110','c':'10','d':'111'}
Soit à décoder **11011101,**
Parmi ces 4 solutions, laquelle correspond à ce code ?

```
['acaaca', 'acaba', 'baaca', 'baba']
```

Le code précédent n'était pas un code préfixe

Changeons encore et utilisons maintenant ce code : code = {'a': '0', 'b': '110', 'c': '10', 'd' : '111'}
Soit à décoder **11001100**
Parmi ces 4 solutions, laquelle correspond à ce code ?

```
['acaaca', 'acaba', 'baaca', 'baba']
```

On a ici à faire à un code préfixe qui est très intéressant dans le cas de code à taille variable.

Coder la fonction qui teste si une chaine s1 est préfix d'une autre chaine s2

```
def est_prefixe(s1,s2):
    n=len(s1)
    return not((n<=len(s2) and s2[:n]==s1))
```

Sauver sous compression.py.

Testons cette fonction :

```
print(est_prefixe('10','110'))
print(est_prefixe('11','110'))
```

Capturer le résultat.

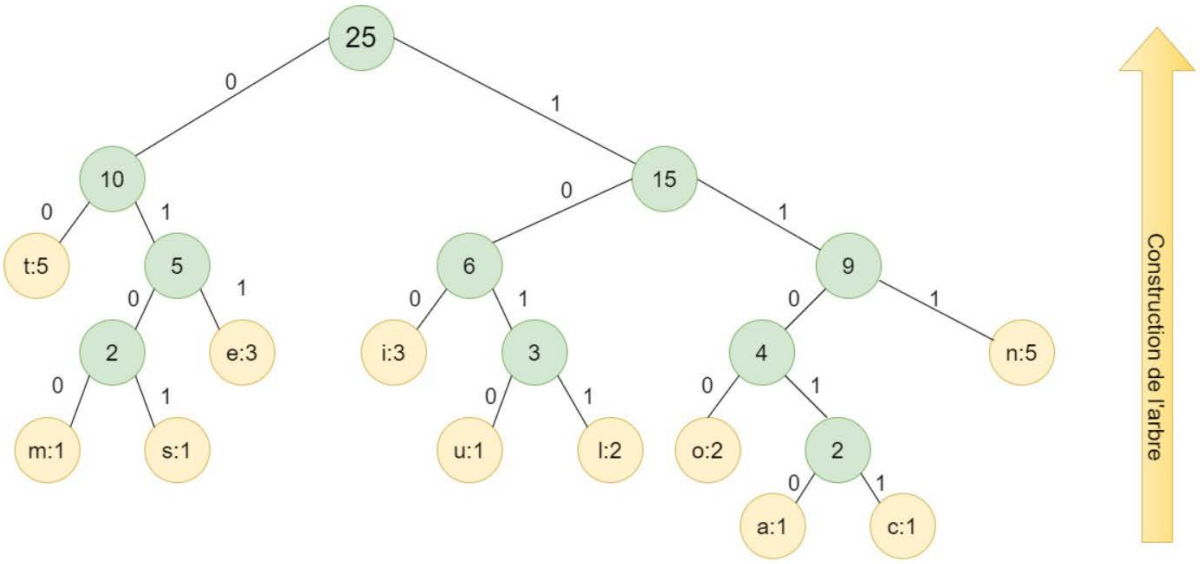
d) Codage de Huffman :

L'algorithme de **Huffman** génère un code-préfixe à taille variable, à partir de la table des fréquences d'une suite de valeurs. Il s'implémente en construisant un arbre binaire de telle sorte que les lettres avec les fréquences les plus élevées soient le plus proches de la racine pour obtenir ***un code binaire le plus petit possible***.

Le code de chaque lettre est obtenu comme suit : il existe un unique chemin qui va de la **racine** à **une feuille donnée**. Chaque fois que l'on descend d'un niveau, on emprunte le fils gauche ou bien le fils droit du nœud où l'on se trouve. On peut ainsi coder le chemin suivi par une liste de 0 et de 1, où 0 signifie "fils gauche" et 1 signifie "fils droit". Le code de la feuille est la concaténation de ces 0 et 1.

Exemple avec le mot : ***anticonstitutionnellement***

Clé	a	c	s	u	m	o	l	i	e	n	t
Fréquence	1	1	1	1	1	2	2	3	3	5	5



On commence par le bas, par les éléments qui sont le moins fréquents ici a et c le nœud aura la valeur 2= fréquence de a + fréquence de c idem pour m et s dont le nœud vaudra aussi 2…..

Donner le code binaire de la lettre n et de la lettre u, le code est-il de type préfixe. ?

Ecrivons la fonction qui donne l'histogramme d'un texte :

```
def histogramme(u):
    tf= {}
    for a in u:
        if a in tf: tf[a] += 1
        else: tf[a] = 1
    return tf
```

Sauver sous compression.py.

Testons cette fonction avec le texte : ***anticonstitutionnellement***

```
code2=histogramme('anticonstitutionnellement')
print("code2",code2)
```

Capturer le résultat.

i. Implémentation de l'algorithme de Huffman

Soit un fichier à compresser :

1. Au préalable, dresser la table des fréquences (Clé=lettres / Valeur=poids)
2. Pour fabriquer l'arbre associé à une table des fréquences on utilise une **file de priorité.** On met dans la file les futures feuilles de notre arbre avec leurs poids comme priorités. Pour cela il suffit de lire la table dont les clés sont les lettres et les valeurs sont les poids.
3. Pour k allant de 0 au nombre de feuilles -2 :

- On retire de la file les deux arbres x et y de priorités minimales.
- On crée l'arbre z dont les deux fils sont x et y .
- On met z dans la file avec la priorité adéquate.

4. Maintenant, la file contient un unique arbre. On renvoie cet arbre.

ii. FILE de priorité

Une **file de priorité** est justement une structure de données capable de maintenir de manière efficace le plus petit élément de la collection d'objets. Quand il est dit « efficacement », il est attendu que l'insertion soit en O(1) et l'extraction soit, au pire, en O(log(n)) où n est le nombre d'éléments.

Deux opérations sont associées à une file de priorité :

- L'insertion d'un nouvel élément (***opération en général désignée par push***) ;
- L'extraction du minimum (***opération en général désignée par pop***) ;

La bibliothèque standard Python propose la structure de données file de priorité via le module **heapq**. Une file de priorité Python accepte n'importe quel type d'objets à condition que Python sache les comparer.

Coder cette file prioritaire :

```
from heapq import *
L=[]
heappush(L,(10,['F','a']))
heappush(L,(8,['F','3']))
heappush(L,(6,['F','20']))
heappush(L,(5,['F','1']))
heappush(L,(4,['F','0']))
print(L)
```

Sauver sous compression.py et capturer le résultat.

```
for x in range(len(L)):
    print(heappop(L))
```

Coder ses deux lignes d'instruction à la suite, sauver, capturer le résultat, qu'obtenez-vous ?

iii. Représenter un code préfixe par un arbre

Maintenant que l'on a défini ce qu'est une liste de priorité on peut continuer pour arriver à la construction de notre arbre binaire en python.

- L'arbre vide est codé par une liste vide [].
- **Si** l'arbre est réduit à une simple feuille contenant la lettre a, nous le représentons par la liste ['F', a] (F comme "feuille", et **a** la valeur de cette clé F du dictionnaire).
- **Sinon,** il possède un fils gauche t1 et un fils droit t2. Soient L et R leurs représentations : nous représentons notre arbre par la liste ['N', L, R] (N comme "nœud").

Coder la fonction permettant de créer l'arbre binaire en partant de l'histogramme du texte ***anticonstitutionnellement .***

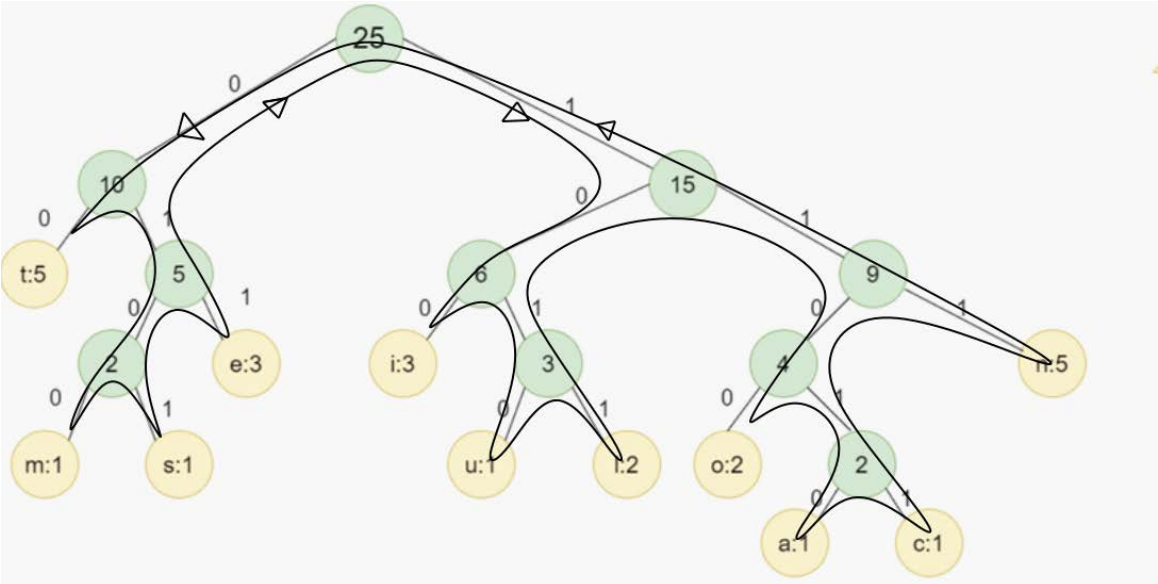
```
def faire_arbre(h): # l'histogramme du texte h est fourni en entrée{'a': 1, 'i': 3, 'u': 1, 'o': 2, 'l': 2, 'm': 1, 'e': 3, 'c': 1, 'n': 5, 's': 1, 't': 5}
    Q=[]    # création d'une liste vide pour y déposer l'ensemble des
            #feuille de l'arbre par priorité contenu dans le l'histogramme

    for a in h: # pour toutes les clés contenues dans l'histogramme
        heappush(Q,(h[a],['F',a])) #entrer les diffférents éléments dans la file de priorité Q en cas de retrait, les éléments retirés sont ceux ayant la priorité la plus faible
                                   #le format entré correspond:  à valeur de la fréquence suivi de  'F',clé

    for k in range (len(h)-1):#Pour k allant de 0 au nombre de feuilles -2:
        px,x=heappop(Q) # On retire de la file les deux arbres x  et y de priorités minimales px contient la valeur correspondant à l'occurence du caractère et x la clé
        py,y=heappop(Q)  # py contient la valeur correspondant à l'occurence du 2 élément et y la clé du 2 element
        z=['N',x,y]     #On crée l'arbre z  dont les deux fils sont x et y .

        heappush (Q,(px+py,z)) #On met l'arbre z en associant la valeur (px+py=somme des 2 occurences) dans la file de priorité .

    return heappop(Q)[1]#on retourne l'arbre z final

t=faire_arbre(code2)
print(t)
```

Sauver sous compression.py et capturer le résultat.

Comment obtenir le code préfixe ?

Une fois l'arbre crée, le code de chaque lettre est obtenu comme suit :

- Il existe un unique chemin qui va de la racine à une feuille donnée.
- **Chaque fois que l'on descend d'un niveau, on emprunte le fils gauche ou bien le fils droit du nœud où l'on se trouve.**
- On peut ainsi coder le chemin suivi par une liste de 0 et de 1, où **0 signifie "fils gauche"** et **1 signifie "fils droit"**.
- **Le code de la feuille est la concaténation de ces 0 et 1.**



Coder la fonction permettant d'obtenir les codes préfixe à partir de l'arbre binaire construit à partir de l'histogramme du texte anticonstitutionnellement

```
def faire_codes(t):

    if t[0] == 'F':    # ['N', ['N', ['F', 't'], ['N', ['N', ['F', 'm'], ['F', 's']]],.....
        return {t[1]:''}   # si la la feuille est trouvée on retourne la clé dans un dictionnaire
                           #avec une valeur associée nulle exemple {'t':''}

    else:

        c1 = faire_codes(t[1]) # tant qu'on a pas de Feuille on parcours recursivement le fils gauche de l'arbre on stocke en memoire la clé du dictionnaire
        print('c1',c1)
        c2 = faire_codes(t[2]) # on parcours recursivement le fils de droit de l'arbre  et fils gauche est recherché par c1
        print('c2',c2)                      #au prochain appel de la fonction jusqu'a trouver une feuille
        print('c1',c1)
        for k in c1:            # on fait précéder chaque code c1 (feuille gauche) par un 0
            c1[k] = '0' + c1[k]
        print(c1)

        for k in c2:            # on fait précéder chaque code c2 (feuille droit) par un 1
            c2[k] = '1' + c2[k]
        print('c2',c2)

        c1.update(c2) # on fusionne les 2 dictionnaires
        print('c1',c1)
        print('on depile')
        return c1

c=faire_codes(t)
print(c)
```

Sauver sous compression.py et capturer le résultat.
Donner les codes préfix obtenus par cette fonction

e) Compression, efficacité :

Copiez la variable texte suivante sous python :
texte="""La Cigale, ayant chanté Tout l'été,
Se trouva fort dépourvue Quand la bise fut venue :
Pas un seul petit morceau De mouche ou de vermisseau.
Elle alla crier famine Chez la Fourmi sa voisine,
La priant de lui prêter Quelques grains pour subsister
Jusqu'à la saison nouvelle. Je vous paierai, lui dit-elle,
Avant l'Août, foi d'animal, Intérêt et principal.
La Fourmi n'est pas prêteuse :
C'est là son moindre défaut.
Que faisiez-vous au temps chaud ?
Dit-elle à cette emprunteuse.

Nuit et jour à tout venant
Je chantais, ne vous déplaise.
Vous chantiez ?
j'en suis fort aise.
Eh bien! dansez maintenant."""

On va maintenant pouvoir comparer selon le type de codages utilisé (ASCII , Huffmann) . La performance du point de vue de la taille occupées en mémoire en nombre de bit.

iv. Commençons par le codage ASCII

Entrez le programme suivant qui va permettre après codage ASCII de donner la longueur en nombre de bit du texte.

```
s1 = coder_ascii(texte)
print('S1',s1)
print("longueur du texte s1 en nombre de bit=",len(s1))
```

Sauver sous compression.py et capturer le résultat.

v. Utilisation du codage de Huffmann

Pour coder le texte selon la méthode de Huffmann réalisons les opérations suivantes :

```
code_texte=histogramme(texte)
print(code_texte)

arbre=faire_arbre(code_texte)


code_prefix=faire_codes(arbre)
print(code_prefix)

s2=(coder(texte,code_prefix))
print('S2',s2)
print('longeur s2=',len(s2))
```

Sauver sous compression.py et capturer le résultat.

En déduire sachant qu'il manque la taille de la table le taux de compression obtenu par rapport au codage ASCII

FIN