Arbres – Exercices de bac

Exercice 1 : *Cet exercice porte sur la programmation objet, la récursivité, les arbres binaires et les systèmes d'exploitation.*
Dans cet exercice, on travaille dans un environnement Linux. On considère l'arborescence de fichiers de la figure 1.

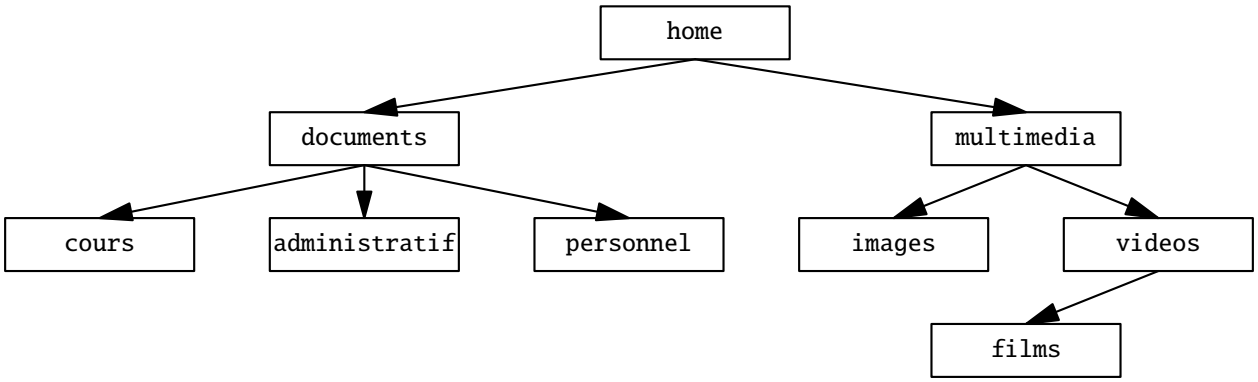


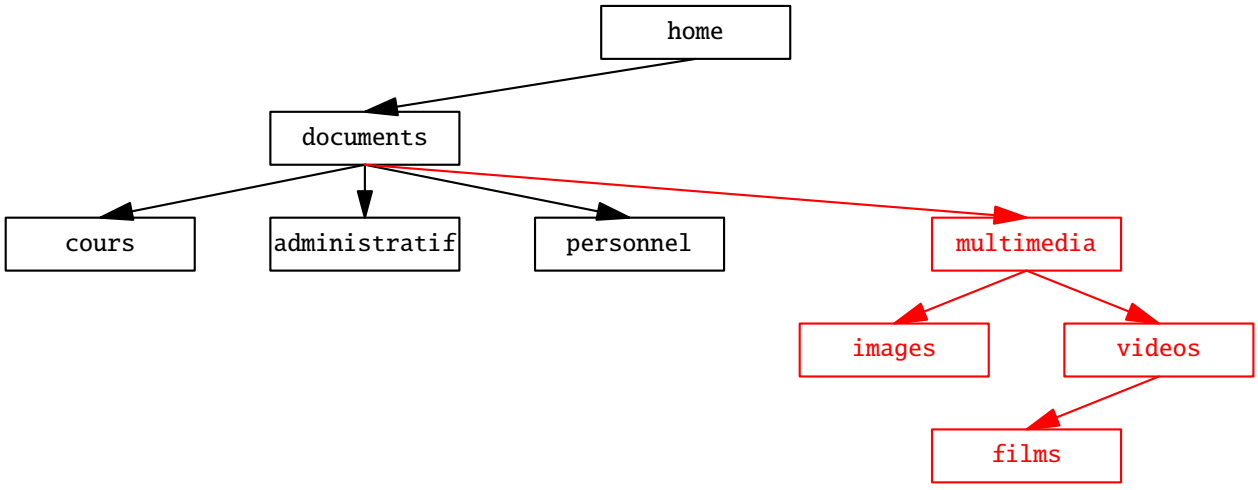
Figure 1. Arborescence de fichiers

Partie A

1) Le répertoire courant est `home`. Donner une commande permettant de connaître le contenu du dossier `documents`.
Solution : Il suffit de faire `ls documents`.

On suppose que l'on se trouve dans le dossier `cours` et que l'on exécute la commande `mv ../../multimedia /home/documents`

2) Indiquer la modification que cela apporte dans l'arborescence de la figure 1.
Solution : Le dossier `multimedia` passe dans `documents`.



On considère le code suivant :

```
class Arbre:
    def __init__(self, nom, g, d):
        self.nom = nom
        self.gauche = g
        self.droit = d
```

```
    def est_vide(self):
        return self.gauche is None and self.droite is None

    def parcours(self):
        print(self.nom)
        if self.gauche != None:
            self.gauche.parcours()
        if self.droit != None:
            self.droit.parcours()
```

3) Donner une raison qui justifie que le code précédent ne permet pas de modéliser l'arborescence de fichiers de la figure 1.
Solution : Cette classe permet de représenter des arbres binaires. Par contre un dossier peut un nombre de sous-dossiers différents de 2, comme `documents`. Cette classe n'est donc pas adaptée.
4) Donner le nom du parcours réalisé par le code précédent.
Solution : On affiche la racine avant de faire les appels récursifs. C'est donc un parcours préfixe.
5) Donner la liste des dossiers dans l'ordre d'un parcours en largeur de l'arborescence. On ne demande pas d'écrire ce parcours en Python.
Solution : On va obtenir `home`, `documents`, `multimedia`, `cours`, `administratif`, `personnel`, `images`, `videos`, `films`.

Partie B

Pour pouvoir modéliser l'arborescence de fichiers de la figure 1, on propose l'implémentation suivante. L'attribut `fils` est une variable de type **`list`** contenant tous les dossiers `fils`. Cette liste est vide dans le cas où le dossier est vide.

```
class Dossier:
    def __init__(self, nom, liste):
        self.nom = nom
        self.fils = liste # liste d'objets de la classe Dossier
```

6) Écrire le code Python d'une méthode `est_vide` qui renvoie **`True`** lorsque le dossier est vide et **`False`** sinon.

```
    def est_vide(self):
        return self.fils == []
```

7) Écrire le code Python permettant d'instancier une variable `var_multimedia` de la classe `Dossier` représentant le dossier `multimedia` de la figure 1. Attention : cela nécessite d'instancier tous les nœuds du sous-arbre de racine `multimedia`.

```
val_films = Dossier("films", [])
val_videos = Dossier("videos", [val_films])
val_images = Dossier("images", [])
var_multimedia = Dossier("multimedia", [var_images, var_videos])
```

8) Recopier et compléter sur votre copie le code Python de la méthode `parcours` suivante qui affiche les noms de tous les descendants d'un dossier en utilisant l'ordre préfixe.

```
def parcours(self):
    print(self.nom)
    for f in self.fils:
        f.parcours()
```

9) Justifier que cette méthode `parcours` termine toujours sur une arborescence de fichiers.
Solution : À chaque étape on descend dans l'arborescence. On finira par arriver sur des dossiers vides, puisqu'il y a un nombre fini de dossiers.

10) Proposer une modification de la méthode `parcours` pour que celle-ci effectue plutôt un parcours suffixe (ou postfixe).
Solution : Il faut afficher le nom du dossier à la fin.

```
def parcours(self):
    for f in self.fils:
        f.parcours()
    print(self.nom)
```

11) Expliquer la différence de comportement entre un appel à la méthode `parcours` de la classe `Dossier` et une exécution de la commande UNIX `ls`.
Solution : La commande `ls` n'est pas récursive et n'affiche que le contenu du dossier en cours ou de celui demandé.

On considère la variable `var_videos` de type Dossier représentant le dossier `videos` de la figure 1. On souhaite que le code Python `var_videos.mkdir("documentaires")` crée un dossier `documentaires` vide dans le dossier `var_videos`.

12) Écrire le code Python de la méthode `mkdir`.

```
def mkdir(self, nom):
    self.fils.append(Dossier(nom, []))
```

13) Écrire en Python une méthode `contient(self, nom_dossier)` qui renvoie `True` si l'arborescence de racine `self` contient au moins un dossier de nom `nom_dossier` et `False` sinon.

```
def contient(self, nom):
    if self.nom == nom:
        return True
    else:
        for f in self.fils:
            if f.contient(nom):
                return True
        return False
```

14) Avec l'implémentation de la classe `Dossier` de cette partie, expliquer comment il serait possible de déterminer le dossier parent d'un dossier donné dans une arborescence donnée. On attend ici l'idée principale de l'algorithme décrite en français. On ne demande pas d'implémenter cet algorithme en Python.
Solution : Il faut parcourir l'arborescence en cherchant un dossier qui contient le dossier actuel. Cela peut être problématique s'il y a plusieurs dossiers de même nom.

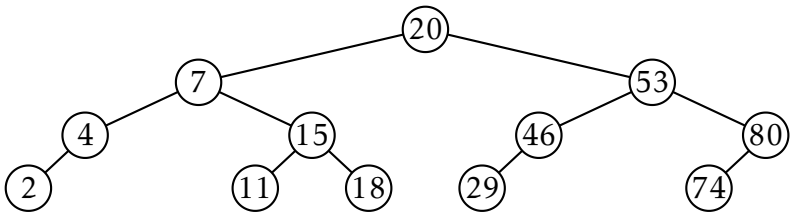
15) Proposer une modification dans la méthode `__init__` de la classe `Dossier` qui permettrait de répondre à la question précédente beaucoup plus efficacement et expliquer votre choix.
Solution : On peut rajouter un attribut `parent` et rajouter un paramètre permettant d'obtenir cette valeur dans les paramètres du constructeur. Il faut aussi modifier `mkdir`.

```python
class Dossier:
    def __init__(self, nom, liste, parent=None):
        self.nom = nom
        self.fils = liste # liste d'objets de la classe Dossier
        self.parent = parent

    def mkdir(self, nom):
        self.fils.append(Dossier(nom, [], self))
```

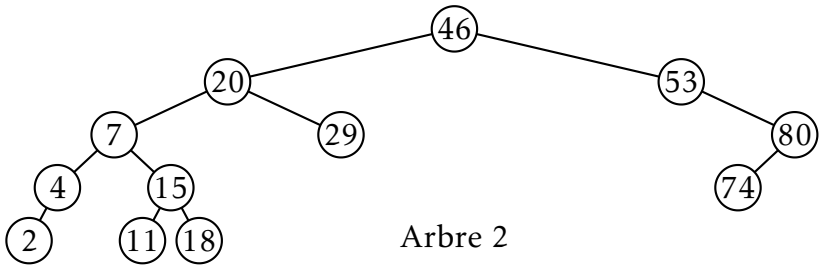
Exercice 2 : *Cet exercice traite des arbres et de l'algorithmique.*
Dans cet exercice, la taille d'un arbre est égale au nombre de ses nœuds et on convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1.
On utilisera la définition suivante : un arbre binaire de recherche est un arbre binaire, dans lequel :

- on peut comparer les valeurs des nœuds : ce sont par exemple des nombres entiers, ou des lettres de l'alphabet ;
- si `x` est un nœud de cet arbre et `y` est un nœud du sous-arbre gauche de `x`, alors il faut que `y.valeur` $<$ `x.valeur`.
- si `x` est un nœud de cet arbre et `y` est un nœud du sous-arbre droit de `x`, alors il faut que `y.valeur` $\geq$ `x.valeur`.

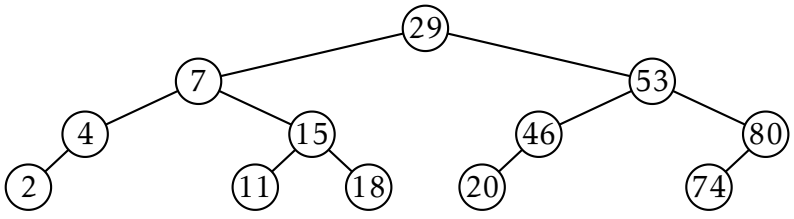
1) Parmi les trois arbres dessinés ci-dessous, entourer celui qui n'est pas un arbre binaire de recherche. Justifier.



Arbre 1



Arbre 2



Arbre 3

Une classe `ABR`, qui implémente une structure d'arbre binaire de recherche, possède l'interface suivante :

```python
class ABR:
    def __init__(self, valeur, sa_gauche, sa_droit):
```

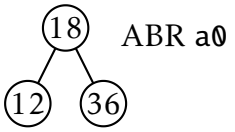
```
        self.valeur = valeur   # valeur de la racine
        self.sa_gauche = sa_gauche   # sous-arbre gauche
        self.sa_droit = sa_droit   # sous-arbre droit

    def inserer_noeud(self, valeur):
        """Renvoie un nouvel ABR avec le nœud de valeur `valeur`
        inséré comme nouvelle feuille à sa position correcte"""
        # code non étudié dans cet exercice
```

On prendra la valeur **None** pour représenter un sous-arbre vide.

2) La construction d'un ABR se fait en insérant progressivement les valeurs à partir de la racine : la méthode `inserer_noeud` (dont le code n'est pas étudié dans cet exercice) place ainsi un nœud à sa "bonne place" comme feuille dans la structure, sans modifier le reste de la structure. On admet que la position de cette feuille est unique.

   a) En utilisant les méthodes de la classe `ABR`:

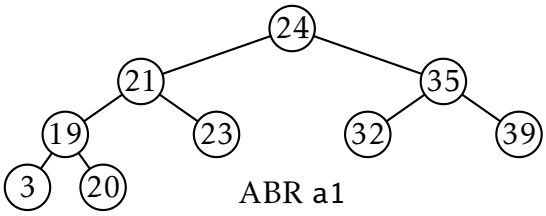
   - écrire l'instruction Python qui permet d'instancier un objet `a0`, de type `ABR`, ayant un seul nœud (la racine) de valeur `18`.
     ```
     a0 = ABR(18, None, None)
     ```
   - écrire une séquence d'instructions qui permet ensuite d'insérer dans l'objet `a0` les deux feuilles de l'arbre de valeurs `12` et `36`.
     ```
     a0.inserer_noeud(12)
     a0.inserer_noeud(36)
     ```



ABR a0

   Selon l'ordre dans lequel les valeurs sont insérées, on construit des ABR ayant des structures différentes.

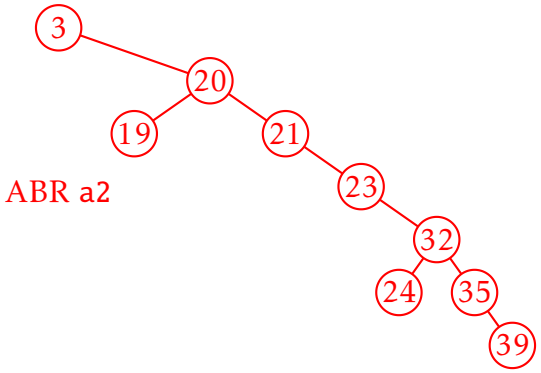
   Voilà par exemple ci-dessous un ABR (nommé a1) obtenu en créant une instance de type ABR ayant un seul nœud (la racine) de valeur 24 puis en insérant successivement les valeurs dans l'ordre suivant : 21; 35; 19; 23; 32; 39; 3; 20



ABR a1

   b) Dessiner l'ABR (nommé a2) que l'on obtiendrait en créant une instance de type `ABR` ayant un seul nœud (la racine) de valeur 3 puis en insérant successivement les valeurs dans l'ordre suivant :

   20; 19; 21; 23; 32; 24; 35; 39



ABR a2

   c) Donner la hauteur des ABR a1 et a2. Hauteur de a1 : 4. Hauteur de a2 : 7

d) On complète la classe ABR avec une méthode `calculer_hauteur` qui renvoie la hauteur de l'arbre.
Compléter les deux commentaires et l'instruction manquante dans le code ci-après de cette méthode.
On pourra utiliser la fonction Python **`max`** qui prend en paramètres deux nombres et renvoie le maximun de ces deux nombres.

```
def calculer_hauteur(self):
    """Renvoie la hauteur de l'arbre"""
    if self.sa_droit is None and self.sa_gauche is None:
        # l'arbre est réduit à une feuille
        return 1
    elif self.sa_droit is None:
        # arbre avec une racine et seulement un sous-arbre gauche
        return 1 + self.sa_gauche.calculer_hauteur()
    elif self.sa_gauche is None:
        # arbre avec une racine et seulement un sous-arbre droit
        return 1 + self.sa_droit.calculer_hauteur()
    else:
        # arbre avec 2 sous-arbres
        return 1 + max(self.sa_gauche.calculer_hauteur(),
                       self.sa_droit.calculer_hauteur())
```

3) La différence de hauteur entre l'ABR `a1` et l'ABR `a2` aura des conséquences lors de la recherche d'une valeur dans l'ABR.

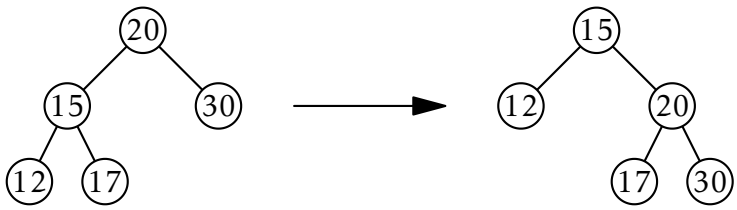
a) Compléter le code ci-dessous de la méthode `rechercher_valeur`, qui permet de tester la présence ou l'absence d'une valeur donnée dans l'ABR :

```
def rechercher_valeur(self, v):
    """Renvoie True si la valeur v
    est trouvée dans l'ABR et False sinon"""
    if v == self.valeur:
        return True
    elif v < self.valeur and self.sa_gauche is not None:
        return self.sa_gauche.rechercher_valeur(v)
    elif v > self.valeur and self.sa_droit is not None:
        return self.sa_droit.rechercher_valeur(v)
    else:
        return False
```

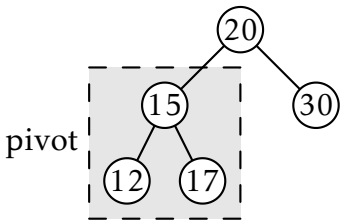
b) On admet que le nombre de fois où la méthode `rechercher_valeur` est appelée pour rechercher la valeur 39 dans l'ABR `a2` est 7.
Donner le nombre de fois où la méthode `rechercher_valeur` est appelée pour rechercher la valeur 20 dans l'ABR `a1`. 4

4) Il existe des algorithmes pour modifier la structure d'un ABR, afin par exemple de diminuer la hauteur d'un ABR. On s'intéresse aux algorithmes appelés **rotation**, consistant à faire "pivoter" une partie de l'arbre autour d'un de ses nœuds.

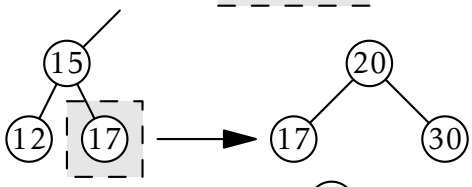
L'exemple suivant permet d'expliquer l'algorithme pour réaliser une rotation droite d'un ABR autour de sa racine :



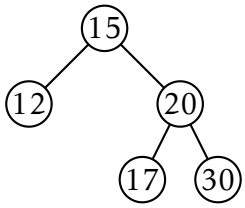
- On appelle **pivot** le sous-arbre gauche de la racine de l'arbre



- Le sous-arbre droit du pivot devient le sous-arbre gauche de la racine



- La racine ainsi modifiée devient le sous-arbre droit du pivot et la racine du pivot devient la nouvelle racine de l'ABR



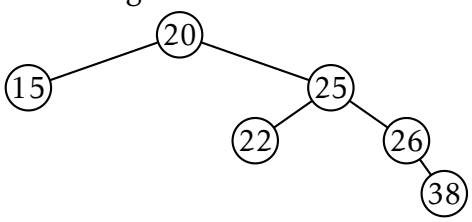
On admet que ces transformations conservent la propriété d'ABR de l'arbre.
La méthode `rotation_droite` ci-après renvoie une nouvelle instance de type `ABR`, correspondant à une rotation droite de l'objet de type `ABR` à partir duquel elle est appelée :

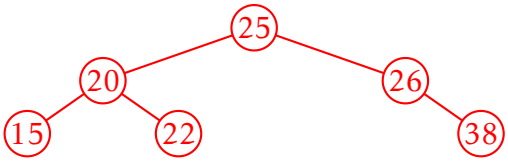
```
def rotation_droite(self):
    """Renvoie une instance d'un ABR après une rotation droite.
    On suppose qu'il existe un sous-arbre gauche"""
    pivot = self.sa_gauche
    self.sa_gauche = pivot.sa_droit
    pivot.sa_droit = self
    return ABR(pivot.valeur, pivot.sa_gauche, pivot.sa_droit)
```

Pour réaliser une rotation gauche, on suivra alors l'algorithme suivant :

- on appelle pivot le sous-arbre droit de la racine de l'arbre,
- le sous-arbre gauche du pivot devient le sous-arbre droit de la racine,
- la racine ainsi modifiée devient le sous-arbre gauche du pivot et la racine du pivot devient la nouvelle racine de l'ABR.

a) En suivant les différentes étapes de cet algorithme, dessiner l'arbre obtenu après une rotation gauche de l'ABR suivant :





b) Écrire le code d'une méthode Python `rotation_gauche` qui réalise la rotation gauche d'un ABR autour de sa racine.

```
def rotation_gauche(self):
    """Renvoie une instance d'un ABR après une rotation gauche.
    On suppose qu'il existe un sous-arbre droit"""
    pivot = self.sa_droit
    self.sa_droit = pivot.sa_gauche
    pivot.sa_gauche = self
    return ABR(pivot.valeur, pivot.sa_gauche, pivot.sa_droit)
```

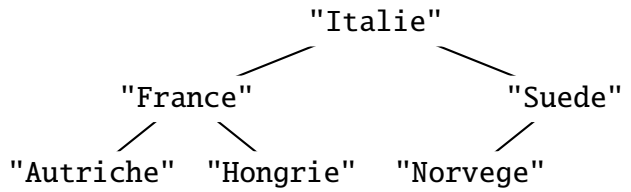
Exercice 3 : *Cet exercice porte sur les arbres binaires de recherche et les algorithmes associés*

Les arbres binaires de recherche considérés ici sont des arbres binaires où les nœuds désignent des chaînes de caractères et pour lesquelles la valeur de chaque nœud est supérieure à celles des nœuds de son enfant gauche, et inférieure à celles des nœuds de sont enfant droit. La relation d'ordre notée < est ici la relation d'ordre alphabétique.

Dans cet exercice, on utilisera la convention suivantes : la hauteur d'un arbre binaire ne comportant qu'un nœud est 1.

Dans cet exercice les arbres binaire de recherche ne contiennent que des noms de pays tous distincts.

On considère l'arbre binaire de recherche ci-contre :

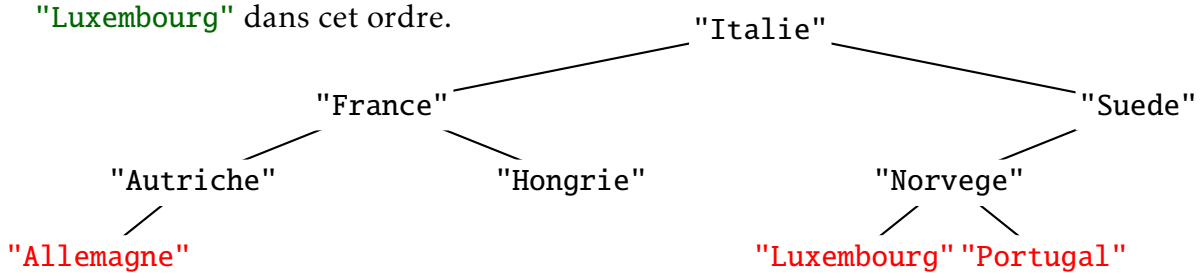


1) a) Donner sans justification la hauteur de cet arbre.
Solution : La hauteur est de 3.

b) Donner sans justification la valeur booléenne de l'expression `"Allemagne" < "Portugal"`.

Solution : `"Allemagne" < "Portugal"` vaut **True**.

c) Compléter l'arbre ci-dessous après l'ajout de `"Allemagne"`, de `"Portugal"` et de `"Luxembourg"` dans cet ordre.

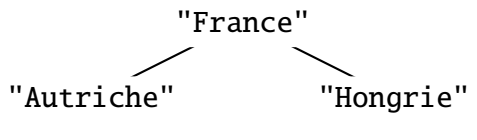


Pour les questions 2, 3 et 4 on traite de l'arbre initial, donc sans l'ajout de `"Allemagne"`, `"Portugal"` et `"Luxembourg"`.

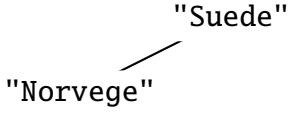
2) On souhaite parcourir l'arbre. Indiquer l'ordre de visite des nœuds lors d'un parcours en largeur. **Solution :** `"Italie"`, `"France"`, `"Suede"`, `"Autriche"`, `"Hongrie"`, `"Norvege"`

3) On souhaite écrire une fonction pour déterminer si le nom d'un pays est dans l'arbre.
On dispose pour cela de :

- la fonction `est_vide` qui prend en paramètre un arbre `arb`. Cette fonction renvoie **True** si l'arbre `arb` est vide, **False** sinon ;
- la fonction `gauche` qui prend en paramètre un arbre `arb` et renvoie son sous-arbre gauche.
Exemple : si `A` est notre arbre initial, `gauche(A)` renvoie l'arbre ci-contre.



- la fonction `droite` qui prend en paramètre un arbre `arb` et renvoie son sous-arbre droit.
Exemple : si `A` est notre arbre initial, `droite(A)` renvoie l'arbre ci-contre.



- la fonction `racine` qui prend en paramètre un arbre `arb` et renvoie la valeur de la racine de l'arbre.
Exemple : `racine(A)` renvoie `"Italie"`.

Recopier, en complétant les lignes 2, 6, 7 et 10, la fonction `recherche` donnée ci-dessous et écrite en Python. Cette fonction prend en paramètre un arbre `arb` et une valeur `val`.

L'appel `recherche(arb, val)` renvoie un booléen (**True** si la valeur `val` est dans l'arbre `arb`, **False** sinon).

```
1  def recherche(arb, val):
2      """ renvoie un booléen indiquant si val est dans arb """
3      if est_vide(arb):
4          return False
5      if val == racine(arb):
6          return True
7      if val < racine(arb):
8          return recherche(gauche(arb), val)
9      else:
10         return recherche(droite(arb), val)
```

4) Écrire une fonction récursive `taille` permettant de déterminer le nombre de pays présents dans un arbre.

Cette fonction prendra en paramètre un arbre `arb` et renverra un entier.

```
def taille(arb):
    if est_vide(arb):
        return 0
    else:
        return 1 + taille(gauche(arb)) + taille(droite(arb))
```

Exercice 4 : Cet exercice porte sur les arbres et la compression d'un fichier texte.

Quand il s'agit de transmettre de l'information sur un canal non bruité, l'objectif prioritaire est de minimiser la taille de la représentation de l'information : c'est le problème de la *compression de données*. Le code de Huffman (1952) est un code de longueur variable optimal, c'est-à-dire tel que la longueur moyenne d'un texte codé est minimale. On observe ainsi des réductions de taille de l'ordre de 20 % à 90 %. Ce code est largement utilisé, souvent combiné avec d'autres méthodes de compression.

Partie A : coder du texte

On donne, en Figure 1 ci-dessous, la table d'encodage hexadécimal des caractères ISO/CEI 8859-1, dite ASCII Latin 1.

Chaque caractère est codé sur 8 bits, soit deux chiffres hexadécimaux, correspondant respectivement à la ligne et à la colonne à l'intersection desquelles il figure.

Par exemple, pour la lettre 'H' figurant à l'intersection de la ligne '4x' et de la colonne 'x8', le code hexadécimal est '48'.

La chaîne de caractère 'Hello_World_!' est codé par :

'48 65 6C 6C 6F 5F 57 6F 72 6C 64 5F 21'

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	positions inutilisées															
1x																
2x	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	positions inutilisées															
9x																
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figure 1. Table ISO/CEI 88-59-1

Dans cette table, le caractère ESPACE est symbolisé par SP.
Soit la chaîne de caractères `txt = "SIX ANANAS"`.

1) Calculer la taille en octets du texte contenu dans la variable `txt`. En déduire la taille en bits nécessaire pour le stocker.
2) Donner le codage de la chaîne de caractères `txt`.

Partie B : Compression de Huffman

Nombre d'occurrences

On appelle nombre d'occurrences d'un symbole le nombre de répétitions de ce symbole dans le texte étudié. Ainsi, dans la phrase "DEECDDEBFACCECCEDBAEE" on peut associer le tableau d'occurrences ci-dessous :

Symbole	A	B	C	D	E	F
Nombre d'occurrences	2	2	5	4	7	1

3) Écrire le tableau d'occurrences associé à la chaîne de caractères `txt`.
4) Préciser à quoi correspond la somme des nombres d'occurrences.

Ce tableau d'occurrences peut être stocké dans un dictionnaire Python où les clés sont les symboles rencontrés dans le texte et les valeurs les nombres d'occurrences de chaque symbole. Ainsi, pour l'exemple ci-dessus, le dictionnaire serait :

```
{'D': 4, 'E': 7, 'C': 5, 'B': 2, 'F': 1, 'A': 2}.
```

5) Recopier et compléter le code de la fonction occurrence ci-dessous qui, pour un texte passé en paramètre, renvoie le dictionnaire d'occurrences associé.

```
def occurrence(texte):
    dico =
    for lettre in ...:
        if lettre in ...:
            dico[lettre] = dico[lettre]+1
        else:
            ...
    return ...
```

Arbre de Huffman

L'algorithme de Huffman met en œuvre plusieurs structures de données. Il opère sur un ensemble dynamique d'arbres binaires pondérés (une *forêt*), structuré en file à priorité.
Initialement, la forêt est constituée d'arbres binaires,

- tous restreints à leur seule racine, dont l'étiquette est un symbole du texte ;
- et respectivement dotés d'un poids correspondant à l'effectif de ce symbole.

Une opération de greffe de deux arbres pondérés est possible : l'arbre résultant est un arbre binaire dont :

- la racine est un nœud sans étiquette ;
- les sous-arbres gauches et droits sont les deux arbres greffés ;
- le poids est la somme des poids de ces deux sous-arbres.

La file à priorité, qui contient tous les arbres considérés, est une structure permettant

- l'extraction : le premier arbre disponible est un arbre de priorité maximale parmi tous les arbres ;
- l'insertion : tout nouvel arbre pondéré est inséré
  - après tous ceux qui ont une priorité strictement plus grande que la sienne ;
  - avant tous ceux qui ont une priorité inférieure ou égale à la sienne.

Pour construire l'arbre de Huffman, tant que la forêt compte au moins deux arbres,

- les deux arbres prioritaires sont extraits de la file ;
- ils sont greffés en un nouvel arbre pondéré ;
- et celui-ci est inséré dans la file à priorité.

Une fois l'arbre construit, on pondère les arêtes en partant de la racine : 0 pour les arêtes menant aux enfants gauches, 1 pour les arêtes menant aux enfants droits.
Le schéma de la figure ci-dessous indique comment on construit un arbre de Huffman en fonction du tableau d'occurrences.
Pour plus de clarté, les étiquettes de tous les nœuds ont été remplacées par le poids de l'arbre dont ils sont la racine.

6) Construire l'arbre de Huffman associé à la chaîne de caractères `txt`.
7) Préciser à quoi correspond le poids de la racine de cet arbre.

Codage et compression à l'aide de l'arbre de Huffman

À l'aide de l'arbre de Huffman, on peut créer une table de codage où chaque symbole est codé par les bits lus sur le chemin entre la racine de l'arbre et la feuille correspondant au symbole. Dans l'exemple ci-dessus, la lettre 'F' serait codée par 0110 et la lettre 'E' par 11.

8) Indiquer le type de parcours à utiliser sur l'arbre de Huffman pour réaliser cette table de codage.
9) Donner la table de codage pour la chaîne de caractères txt.
10) Justifier le fait que le code de Huffman est un code de longueur variable.

Le codage du texte se fait ensuite caractère par caractère en utilisant la table de codage.

11) Coder la chaîne de caractères txt à l'aide du code de Huffman et de l'arbre construit à la question 6.
12) En reprenant le résultat déterminé dans la partie A, en déduire le taux de compression en % pour la variable txt et vérifier l'assertion du texte d'introduction : "On observe ainsi des réductions de taille de l'ordre de 20 % à 90 %."

Le taux de compression est le ratio : $\dfrac{\text{encombrement initial - encombrement final}}{\text{encombrement initial}}$

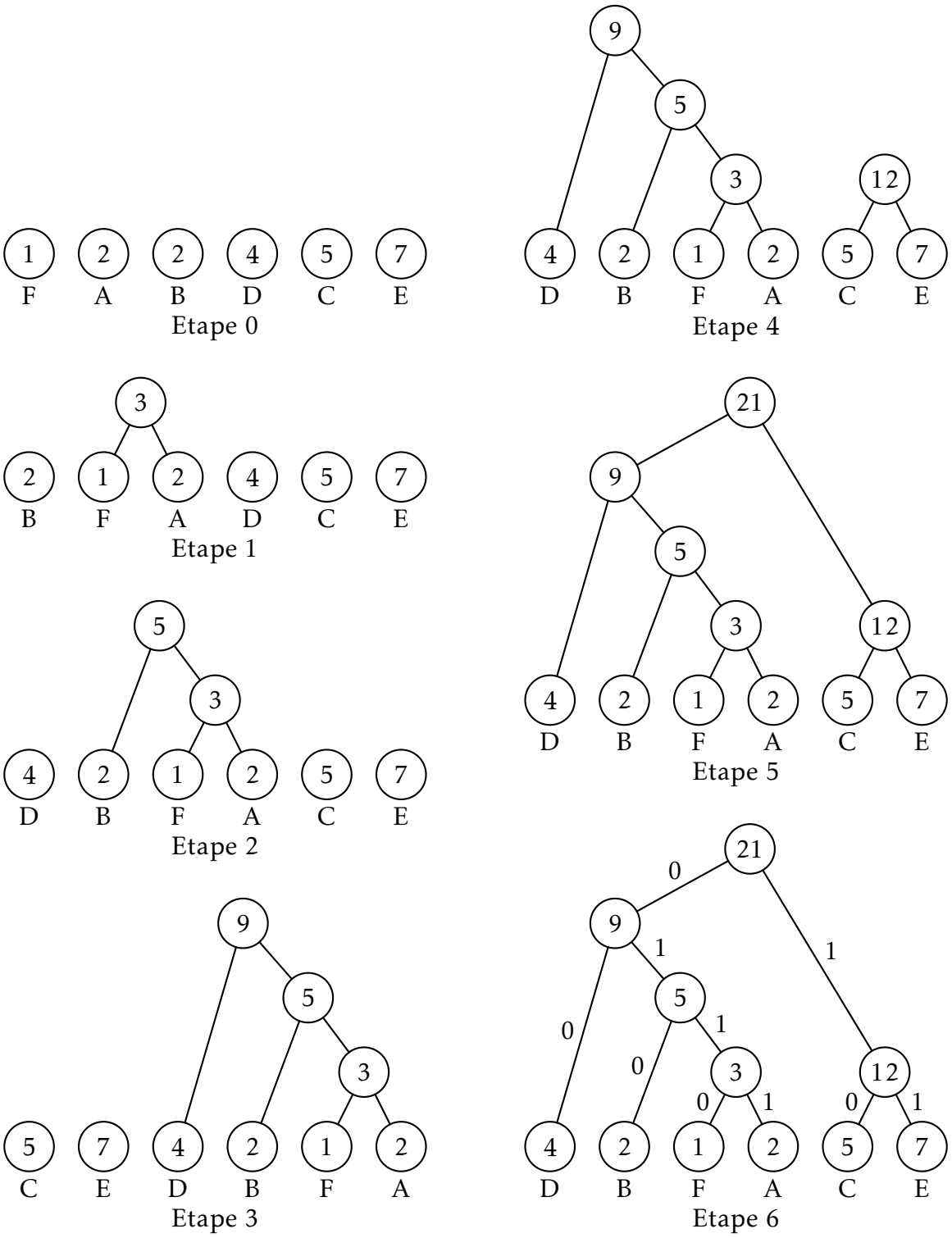


Figure 2. Construction de l'arbre de Huffman