Autotest n°4 – Correction

Exercice 1 : *Cet exercice porte sur la programmation Python, la programmation orientée objet, les bases de données relationnelles et les requêtes SQL.*

Partie A

Une entreprise, présente sur différents sites en France, attribue à chacun de ses employés un numéro de badge unique.
Dans le tableau ci-dessous, on donne le numéro de badge, le nom, le prénom et les années de naissance et d'entrée dans l'entreprise de quelques salariés.

numéro badge	nom	prénom	année de naissance	année d'entrée
112	LESIEUR	Isabelle	1982	2005
2122	VASSEUR	Adrien	1962	1980
135	HADJI	Hakim	1992	2015

Pour chaque personne, on souhaite stocker les informations dans un objet de la classe `Personne` définie ci-dessous :

```
class Personne():
    def __init__(self, num, n, p, a_naiss, a_entree):
        self.num_badge = num
        self.nom = n
        self.prenom = p
        self.annee_naissance = a_naiss
        self.annee_entree = a_entree
```

1) Écrire à l'aide du tableau précédent, l'instruction permettant de créer l'objet `personneA` de la première personne du tableau : LESIEUR Isabelle.
Solution : `personneA = Personne(112, 'LESIEUR', 'Isabelle', 1982, 2005)`
2) Donner l'instruction permettant d'obtenir le numéro de badge de l'objet `personneA` instancié à la question précédente.
Solution : `personneA.num_badge`

On souhaite ajouter une méthode `annee_anciennete` à la classe Personne qui donne le nombre d'années d'ancienneté d'une personne au sein de l'entreprise. Par exemple : Madame LESIEUR Isabelle a une ancienneté dans l'entreprise de 19 ans en considérant que nous sommes en 2024.

3) Compléter le code suivant de la méthode `annee_anciennete` :

```
    def annee_anciennete(self):
        return 2024 - self.annee_entree
```

On considère la classe `Personnel` qui modélise la liste du personnel d'une entreprise et dont le début de l'implémentation est la suivante :

```
class Personnel:
    def __init__(self):
        self.liste = []
```

4) Écrire la méthode `ajouter` permettant d'ajouter un objet de type `Personne` à la liste du personnel de l'entreprise de la classe `Personnel`.

```
    def ajouter(self, p):
        self.liste.append(p)
```

5) Écrire la méthode `effectif` de la classe `Personnel`. Cette méthode devra renvoyer le nombre de personnes présentes dans l'entreprise.

```
    def effectif(self):
        return len(self.liste)
```

6) Compléter la méthode `donne_nom` de la classe `Personnel`. Cette méthode prend en paramètre le numéro de badge d'une personne et renvoie le nom de la personne correspondant à ce badge si elle existe, ou `None` sinon.

```
def donne_nom(self, num):
    for elt in self.liste:
        if num == elt.num_badge:
            return elt.nom
    return None
```

7) Lors de la célèbre cérémonie des vœux, l'entreprise souhaite mettre à l'honneur les personnes ayant exactement 10 ans d'ancienneté dans l'entreprise. Écrire une méthode de la classe `Personnel` `nb_personne_honneur` qui prend en paramètre l'année de la cérémonie et qui retourne le nombre de personne(s) à mettre à l'honneur.

```
    def nb_personne_honneur(self, annee):
        n = 0
        for elt in self.liste:
            if annee - elf.annee_entree == 10:
                n = n + 1
        return n
```

8) Écrire une méthode `plus_anciens` de la classe `Personnel` qui retourne la liste des numéros de badge des personnes ayant la plus grande ancienneté dans l'entreprise.

```
    def plus_anciens(self):
        res = []
        maxi = 0
        for elt in self.liste:
            anc = elt.annee_anciennete()
            if anc > maxi:
                res = [elt.num_badge]
                maxi = anc
            elif anc == maxi:
                res.append(elt.num_badge)
        return res
```

Partie B

On utilise maintenant une base de données relationnelle. La table `Personnel` dont un extrait est donné ci-dessous contient toutes les données importantes sur le personnel de l'entreprise. L'attribut `num_centre` désigne le numéro du centre dans lequel travaille une personne.

Table `Personnel`

num_badge	nom	prenom	num_centre	annee_naiss	annee_debut
112	LESIEUR	Isabelle	1	1982	2005
2122	VASSEUR	Adrien	2	1962	1980
135	HADJI	Hakim	1	1992	2015

L'attribut `num_badge` est la clé primaire pour la table `Personnel`.

```
SELECT nom, prenom
FROM Personnel
WHERE num_centre = 2;
```

9) Décrire par une phrase en français le résultat de la requête SQL ci-contre.

Solution : On obtient le nom et le prénom de tous les personnels du centre numéro 2.

10) Monsieur HADJI Hakim vient d'obtenir une mutation pour le centre numéro 3. Donner la requête permettant de modifier son numéro de centre sachant que son numéro de badge est 135.

```
UPDATE Personnel SET num_centre = 3 WHERE num_badge = 135;
```

On souhaite proposer plus d'informations sur les différents centres de l'entreprise. Pour cela, on crée une deuxième table `Centre` avec les attributs suivants :

- `num` de type INT ;
- `nom` de type TEXT ;
- `num_tel` de type TEXT ;
- `ville` de type TEXT.

Table `Centre`			
`num`	`nom`	`num_tel`	`ville`
1	Normandie	0450646859	Caen
2	PACA	0450646859	Marseille

11) Expliquer l'intérêt d'utiliser deux tables (`Personnel` et `Centre`) au lieu de regrouper toutes les informations dans une seule table.
Solution : Cela permet d'éviter la redondance des données. Par exemple, si on change le numéro de téléphone d'un centre, on n'a pas besoin de le faire pour tous les employés de ce centre. Il suffit de changer le numéro dans la table `Centre`.

12) Expliquer comment les tables `Centre` et `Personnel` sont mises en relation.
Solution : La clé primaire `num` de `Centre` est aussi la clé étrangère `num_centre` de `Personnel`. Cela permet de relier les deux tables.

13) Écrire une requête permettant d'avoir les noms des personnes travaillant dans le centre de Lille et ayant été embauchées entre 2015 (inclus) et 2020 (inclus).

```
SELECT Personnel.nom FROM Personnel
JOIN Centre ON Centre.num = Personnel.num_centre
WHERE ville = 'Lille' AND 2015 <= annee_debut <= 2020;
```

Le centre de Normandie vient d'être fermé, mais les personnes de ce centre n'ont pas encore été affectées dans leur nouveau centre. On souhaite mettre à jour la table `Centre` en premier à l'aide de la requête suivante.

```
DELETE *
FROM Centre
WHERE nom = 'Normandie';
```

14) Expliquer pourquoi cette requête a renvoyé une erreur.

Solution : Puisque la clé primaire de `Centre` est aussi une clé étrangère de `Personnel`, on ne peut pas supprimer d'éléments de `Centre` tant que toutes ses mentions dans `Personnel` n'ont pas été supprimées.

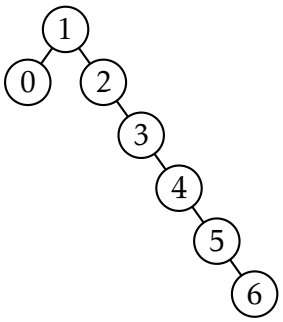
Exercice 2 : *Cet exercice porte sur les arbres binaires de recherche, la POO et la récursivité.*
Nous disposons d'une classe `ABR` pour les arbres binaires de recherche dont les clés sont des entiers :

```
class ABR():
    def __init__(self) :
        # Initialise une instance d'ABR vide.

    def cle(self):
        # Renvoie la clé de la racine de l'instance d'ABR.

    def sad(self):
        # Renvoie le sous-arbre droit de l'instance d'ABR.

    def sag(self):
        # Renvoie le sous-arbre gauche de l'instance d'ABR.

    def est_vide(self):
        # Renvoie True si l'instance d'ABR est vide et False sinon.
```

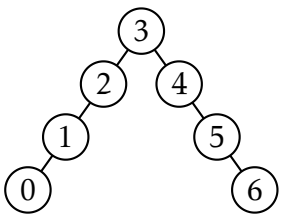
```
    def inserer(self, cle_a_inserer):
        # Insère cle_a_inserer à sa place dans l'instance d'ABR.
```

Considérons ci-dessous trois arbres binaires de recherche :

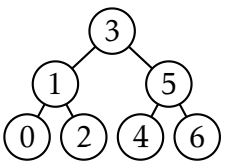
Arbre n°1



Arbre n°2



Arbre n°3



Dans tout l'exercice, nous ferons référence à ces trois arbres binaires de recherche et utiliserons la classe `ABR` et ses méthodes.

Partie A

1) Un arbre est une structure de données hiérarchique dont chaque élément est un nœud. Compléter le texte ci-dessous en choisissant des expressions parmi `au maximum`, `au minimum`, `exactement`, `feuille`, `racine`, `sous-arbre gauche` et `sous-arbre droit` :
   - Le nœud initial est appelé racine
   - Un nœud qui n'a pas de fils est appelé feuille
   - Un arbre binaire est un arbre dans lequel chaque nœud a exactement deux fils.
   - Un arbre binaire de recherche est un arbre binaire dans lequel tout nœud est associé à une clé qui est :
     - supérieure à chaque clé de tous les nœuds de son sous-arbre gauche
     - inférieure à chaque clé de tous les nœuds de son sous-arbre droit
2) Donner dans l'ordre les clés obtenues lors du parcours préfixe de l'arbre n°1.
   Solution : 1, 0, 2, 3, 4, 5, 6.
3) Donner dans l'ordre, les clés obtenues lors du parcours suffixe, également appelé postfixe, de l'arbre n°2.
   Solution : 0, 1, 2, 6, 5, 4, 3.
4) Donner dans l'ordre, les clés obtenues lors du parcours infixe de l'arbre n°3.
   Solution : 0, 1, 2, 3, 4, 5, 6.
5) Compléter les instructions ci-dessous afin de définir puis de construire, en y insérant les clés dans un ordre correct (il y a plusieurs possibilités, on en demande une) , les trois instances de la classe `ABR` qui correspondent aux trois arbres binaires de recherche représentés plus haut.

```
arbre_no1 = ABR()
arbre_no2 = ABR()
arbre_no3 = ABR()
for cle_a_inserer in [1, 0, 2, 3, 4, 5, 6]:
    arbre_no1.inserer(cle_a_inserer)
for cle_a_inserer in [3, 2, 4, 1, 5, 0, 6]:
    arbre_no2.inserer(cle_a_inserer)
for cle_a_inserer in [3, 1, 5, 0, 2, 4, 5]:
    arbre_no3.inserer(cle_a_inserer)
```

6) Voici le code de la méthode `hauteur` de la classe `ABR` qui renvoie la hauteur d'une instance d'`ABR` :

```
def hauteur(self):
    if self.est_vide() :
        return -1
    else :
        return 1 + max(self.sag().hauteur(), self.sad().hauteur())
```

Donner, en vous basant sur cette fonction, la hauteur des trois instances `arbre_no1`, `arbre_no2` et `arbre_no3` de la classe `ABR` définies plus haut et qui correspondent aux trois arbres représentés plus haut.

Solution : La hauteur de `arbre_no1` est de 5, celle de `arbre_no2` est de 3 et celle de `arbre_no3` est de 2.

7) Compléter le code de la méthode `est_presente` ci-contre qui renvoie `True` si la clé `cle_a_rechercher` est présente dans l'instance d'`ABR` et `False` sinon :

```
def est_presente(self, cle_a_rechercher):
    if self.est_vide() :
        return False
    elif cle_a_rechercher == self.cle() :
        return True
    elif cle_a_rechercher < self.cle() :
        return self.sag().est_presente(cle_a_recher
    else :
        return self.sad().est_presente(cle_a_recher
```

8) Expliquer quelle instruction, parmi les trois ci-dessous, nécessitera le moins d'appels récursifs avant de renvoyer son résultat :

- `arbre_no2.est_presente(7)`
- `arbre_no1.est_presente(7)`
- `arbre_no3.est_presente(7)`

Solution : Puisque 7 va s'insérer à droite de 6, c'est dans l'arbre n°3 qu'il y aura le moins d'appels récursifs puisque c'est l'arbre où il est le plus proche de la racine.

Partie B

9) On rappelle que la fonction `abs(x)` renvoie la valeur absolue de `x`. Par exemple :

```
>>> abs(3)
3
```

```
>>> abs(-2)
2
```

On donne la méthode `est_partiellement_equilibre(self)` de la classe `ABR`. Cette méthode renvoie `True` si l'instance de la classe `ABR` est l'implémentation d'un arbre partiellement équilibré et `False` sinon :

```
def est_partiellement_equilibre(self) :
    if self.est_vide() :
        return True
    return abs(self.sag().hauteur() - self.sad().hauteur()) <= 1
```

D'après cette fonction, expliquer ce qu'on appelle ici un arbre *partiellement équilibré*.

Solution : Un arbre est partiellement équilibré s'il est vide ou si la différence entre la hauteur de son sous-arbre gauche et celle de son sous-arbre droit est au plus de 1.

Un arbre binaire est *équilibré* s'il est partiellement équilibré et si ses deux sous- arbres, droit et gauche, sont eux-mêmes équilibrés. Un arbre vide est considéré comme équilibré.

10) Justifier que, parmi les trois arbres définis plus haut, deux sont partiellement équilibrés.

Solution : Dans l'arbre n°1, la différence de hauteur entre les deux sous-arbres est de 4, pour l'arbre n°2, c'est 0 et pour l'arbre n°3 c'est également 0. Les arbres n°2 et n°3 sont donc partiellement équilibrés.

11) Justifier que, parmi les trois arbres définis plus haut, un seul est équilibré.
Solution : Seul l'arbre n°3 est équilibré puisque dans l'arbre n°2, les sous arbres gauches et droites ne sont pas partiellements équilibrés.

12) Définir et coder la méthode récursive `est_equilibre` de la classe `ABR` qui renvoie **`True`** si l'instance de la classe `ABR` est l'implémentation d'un arbre équilibré et **`False`** sinon.

```
    def est_equilibre(self):
        if self.est_vide() :
            return True
        elif self.est_partiellement_equilibre():
            return self.sag().est_equilibre() and self.sad().est_equilibre()
        else:
            return False
```