Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI T^le Année **2025-26**

Devoir surveillé n°3 – Correction

Nom et prénom :

Exercice 1 : (12pt) *Cet exercice porte sur les bases de données relationnelles et les requêtes SQL.*
Dans cet exercice, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`) et `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY`.

La ville de Bois-Plage a décidé d'organiser, pendant un mois de juillet, un tournoi sportif de volley-ball par équipes de 4. Elle met à disposition des personnes intéressées un site d'inscription en ligne qui utilise un système de gestion de base de données.
Le schéma de la base de données utilisée est donné ci-dessous, en figure 1. Sur ce schéma, les clés primaires ont été soulignées et les clés étrangères indiquées par un croisillon (symbole #).

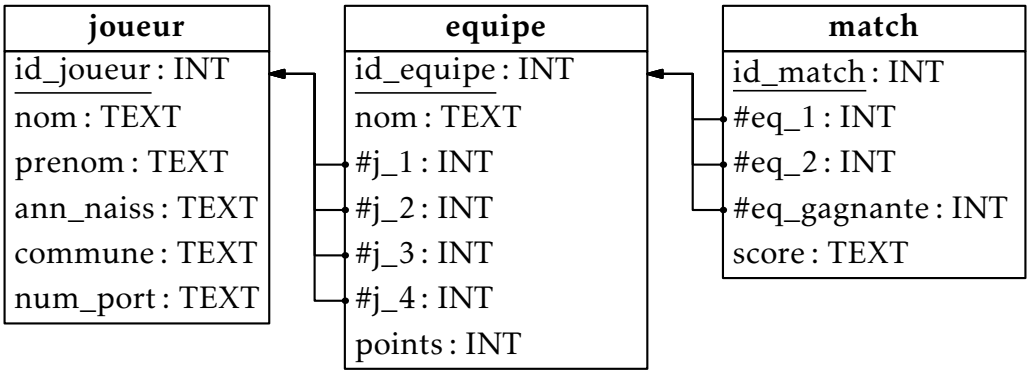


Figure 1. Schéma de la base de données

À la clôture des inscriptions, de nombreuses équipes sont inscrites. On donne ci-dessous des extraits des tables `joueur` et `equipe` obtenues à l'issue de la phase d'inscription.

equipe						
id_equipe	nom	j_1	j_2	j_3	j_4	points
8	Les Mr Freeze	7	12	5	33	0
9	Tagadas Winners	45	23	67	65	0
10	Volley Warriors	25	27	30	35	0
11	Les Piafs	37	32	41	28	0

joueur					
id_joueur	nom	prenom	ann_naiss	commune	num_port
25	Leclerc	Océane	2008	Bois-Plage	0660358945
26	Renault	Henri	1971	Guilland	0625597427
27	Desousa	Laure	1980	Bois-Plage	0746881113
28	Hernand	Yves	1986	Lebrundan	0739401689
29	Giraud	Brigitte	1972	Saint-Adrien	0651936319
30	Barbier	Laure	1979	Bois-Plage	0787028125

1) Expliquer, dans les relations précédentes, le rôle des clés primaires. **Solution :** Les clés primaires servent à rendre unique chacune des entrées de chaque table.
2) Expliquer quelle situation ne serait pas possible dans la table `match` si le champ `id_match` n'avait pas été introduit dans cette table. **Solution :** Sans `id_match`, il ne serait pas possible d'avoir deux matchs avec les mêmes équipes et le même score.

3) Donner le résultat de la requête suivante en l'appliquant à l'extrait de la table joueur donné dans l'énoncé.

```
SELECT prenom FROM joueur WHERE ann_naiss < 1985;
```

Solution : On va obtenir les prénom des joueurs nés avant 1985. On obtiendra donc Henri, Laure, Brigitte et Laure.

4) Modifier la requête précédente afin d'éviter les éventuels doublons.
Solution :
```
SELECT DISTINCT prenom FROM joueur WHERE ann_naiss < 1985;
```

5) Écrire une requête SQL permettant d'obtenir tous les noms, années de naissance et numéros de téléphone portable des personnes qui habitent à Bois-Plage.
Solution :
```
SELECT nom, ann_naiss, num_port FROM joueur WHERE commune = "Bois-Plage";
```

L'organisateur souhaite obtenir l'identité du premier joueur de l'équipe `"les Kangourous"`. L'équipe `"les Kangourous"` n'apparaît pas dans l'extrait.

6) Écrire une requête SQL permettant d'obtenir le nom et le prénom du joueur `j_1` de l'équipe `"les Kangourous"`.

```
SELECT joueur.nom, prenom FROM joueur
JOIN equipe ON equipe.j_1=joueur.id_joueur
WHERE equipe.nom="les Kangourous";
```

L'équipe "Volley Warriors" a terminé le tournoi avec un total de 5 points.

7) Écrire une requête SQL permettant de mettre à jour la table `equipe` avec le nombre de points gagnés par l'équipe `"Volley Warriors"`.
Solution :
```
UPDATE equipe SET points=5 WHERE nom="Volley Warriors";
```

8) Écrire une requête SQL permettant de supprimer de la table `joueur` le joueur ayant pour identifiant le numéro 35.
Solution :
```
DELETE FROM joueur WHERE id_joueur=35;
```

À la clôture du tournoi, la table match est totalement complétée. Un extrait de cette table est donné ci-dessous.

match				
id_match	eq_1	eq_2	eq_gagnante	score
32	3	8	8	25-20
33	3	9	3	25-15
34	3	10	10	25-7

9) Écrire une requête SQL permettant d'obtenir la liste des identifiants de matchs auxquels a participé l'équipe ayant pour identifiant 12.

```
SELECT id_match FROM match
WHERE eq_1=12 OR eq_2=12;
```

10) Écrire une requête SQL permettant d'obtenir la liste des identifiants des matchs pour lesquels le joueur 1 de l'équipe 1 du match vient de la commune de Bois-Plage.

```
SELECT id_match FROM match
JOIN equipe ON match.eq_1=equipe.id_equipe
JOIN joueur ON joueur.id_joueur=equipe.j_1
WHERE commune="Bois-Plage";
```

11) Écrire une requête SQL permettant d'obtenir la liste, classée par ordre alphabétique, des noms et prénoms des joueurs ayant gagné au moins un match en tant que joueur 1 de l'équipe 1 du match.

```
SELECT DISTINCT joueur.nom, prenom FROM joueur
JOIN equipe ON joueur.id_joueur=equipe.j_1
JOIN match ON match.eq_1=equipe.id_equipe
WHERE match.eq_gagnante=match.eq_1
ORDER BY joueur.nom, prenom;
```

Exercice 2 : (14pt) *Cet exercice porte sur les piles, la programmation objet et l'algorithmique.*
Défi Tubes est un jeu à un joueur. Le joueur dispose de 4 tubes. Chaque tube peut contenir de 0 à 3 phases. Chaque phase possède une couleur. Il y a 3 couleurs possibles. On peut s'imaginer ces phases comme des palets de couleur dans le tube.

Pour modéliser les couleurs, on utilisera les entiers 1, 2 et 3. Lorsqu'un tube contient 0 phase, on dit que le tube est vide. Lorsqu'il en a 3, on dit qu'il est plein. Lorsqu'un tube n'est pas vide, sa **dernière** couleur est la couleur de sa phase supérieure.

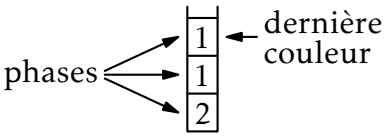


Figure 1. Exemple de tube.

Le jeu **Défi Tube** consiste à verser successivement la dernière couleur des tubes dans les autres tubes avec les contraintes suivantes :

- on ne peut rien verser dans un tube plein ;
- pour verser un **tube 1** dans un **tube 2**, il faut que la dernière couleur du **tube 1** soit la même que celle du **tube 2** ou que le **tube 2** soit vide. Dans ces deux cas, on retire la dernière couleur du **tube 1** pour qu'elle devienne la dernière couleur du **tube 2**. On réitère cela tant que la dernière couleur du **tube 1** est la même et que le **tube 2** n'est pas plein.

Le jeu se termine lorsque 3 des 4 tubes sont pleins et que leurs 3 phases sont de même couleur.
Les figures 2, 3, 4 et 5 ci-après représentent un exemple de partie du jeu **Défi Tube**.

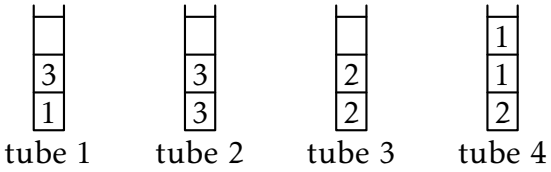


Figure 2. État initial du jeu.

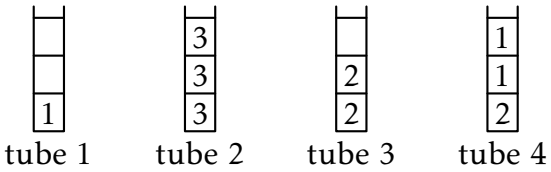


Figure 3. On verse le tube 1 dans le tube 2.

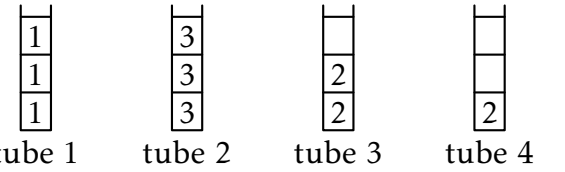


Figure 4. On verse le tube 4 dans le tube 1.

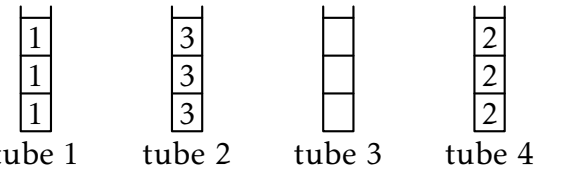
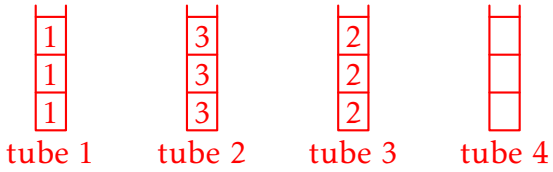


Figure 5. On verse le tube 3 dans le tube 4.

À la figure 5, la partie est terminée.

1) Donner un exemple d'une autre séquence de versements qui aurait permis de terminer le jeu en partant de la situation de la figure 4.
Solution : Il suffit de verser le tube 4 dans le tube 3 :



Ainsi le déroulement du jeu n'est pas unique.

Partie A : Les tubes

Pour modéliser le jeu **Défi Tube**, chaque tube sera représenté par une pile finie de taille maximale 3. Les tubes sont modélisés par des objets de la classe `tube` dont le code est donné ci-dessous.

```
1  class tube:
2      def __init__(self):
3          self.taille = 0
4          self.contenu = [0, 0, 0]
5
6      def est_vide(self):
7          return self.taille == 0
8
9      def empiler(self, couleur):
10         if self.taille < 3:
11             self.contenu[self.taille] = couleur
12             self.taille = self.taille + 1
13
14     def depiler(self):
15         if self.taille > 0:
16             self.taille = self.taille - 1
17             couleur = self.contenu[.........................]
18             self.contenu[self.taille] = 0
19             return ...
20         else:
21             return ...
```

Chaque instance de la classe `tube` a deux attributs :

- l'attribut `taille` représente le nombre d'éléments non nuls dans le tube ;
- l'attribut `contenu` représente la liste (de taille 3) des éléments du tube. Lorsqu'une phase n'est pas vide, elle contiendra une couleur 1, 2, ou 3. Lorsqu'une phase est vide, sa valeur est 0.

Par exemple, le tube ci-contre sera modélisé avec la classe `tube` par le code :

```
t = tube()
t.taille = 2
t.contenu = [1, 3, 0]
```

Figure 6. tube 1

2) Expliquer ce qu'est la structure de pile en précisant à quoi servent les méthodes `empiler` et `depiler`.

Solution : Une pile sert à stocker des données selon le principe dernier arrivé, premier sorti. Les méthodes `empiler` et `depiler` servent à ajouter et enlever des éléments sur cette pile.

3) Expliquer les lignes 11 et 12 du code de la classe `tube`.

Solution : La ligne 11 sert à mettre la couleur au sommet de la pile et la ligne 12 sert à augmenter la taille de cette pile.

4) Compléter le code de la méthode `depiler` précédente. Lorsque le tube est vide, la méthode `depiler` doit renvoyer -1.

Solution :

```
    def depiler(self):
        if self.taille > 0:
            self.taille = self.taille - 1
            couleur = self.contenu[self.taille]
            self.contenu[self.taille] = 0
            return couleur
        else:
            return -1
```

5) Écrire une méthode `est_plein` de la classe `tube`. Cette méthode renvoie **`True`** si le tube est plein et **`False`** si le tube n'est pas plein.

```
    def est_plein(self):
        return self.taille == 3
```

6) Écrire une méthode `est_homogene` de la classe `tube` qui renvoie **`True`** si le tube est plein et si son contenu est composé de trois fois la même couleur, et qui renvoie **`False`** sinon.

```
    def est_homogene(self):
        return (self.est_plein()
                and self.contenu[0] == self.contenu[1]
                and self.contenu[0] == self.contenu[2])
```

7) Écrire une méthode `derniere_couleur` de la classe `tube` qui renvoie le numéro de la dernière couleur du tube. Si le tube est vide, la méthode renverra la valeur -1.

```
    def derniere_couleur(self):
        if self.est_vide():
            return -1
        else:
            return self.contenu[self.taille-1]
```

Le code incomplet d'une méthode verser de la classe tube est donné ci-dessous :

```
def verser(self, other):
    while




        couleur = self.depiler()
        other.empiler(couleur)
```

8) Compléter le code de cette méthode `verser` afin de verser l'instance **`self`** de la classe `tube` dans l'instance `other`. On veillera à vérifier toutes les conditions nécessaires au bon déroulement de cette opération.

```
    def verser(self, other):
        while (not self.est_vide()
            and not other.est_plein()
            and (other.est_vide()
                or self.derniere_couleur() == other.derniere_couleur())):
            couleur = self.depiler()
            other.empiler(couleur)
```

Partie B : Le jeu

Pour modéliser le jeu, on appellera `état` du jeu une liste de 4 tubes. Le code suivant permet de représenter l'`état` de la figure 2.

```
tube1 = tube()
tube1.contenu = [1, 3, 0]
tube1.taille = 2
tube2 = tube()
tube2.contenu = [3, 3, 0]
tube2.taille = 2
tube3 = tube()
tube3.contenu = [2, 2, 0]
tube3.taille = 2
tube4 = tube()
tube4.contenu = [2, 1, 1]
tube4.taille = 3
etat = [tube1, tube2, tube3, tube4]
```

9) En utilisant la méthode `verser` et la variable `etat` représentant la figure 2, écrire un code permettant de faire passer la variable `etat` de la représentation en figure 2 à celle de la figure 3.

```
etat[0].verser(etat[1])  # tube1.verser(tube2)
```

10) Écrire une fonction `gagne` qui prend comme argument un état et qui renvoie **`True`** si la partie est terminée et **`False`** sinon.

```
def gagne(etat):
    nb_tubes_pleins = 0
    for t in etat:
        if t.est_homogene():
            nb_tubes_pleins += 1
    return nb_tubes_pleins == 3
```