Métropole - juin 2024 - sujet 1 (corrigé)

Exercice 1 (POO et graphes)

1. Puisqu'aucun site ne pointe sur le **site2**, `s2` n'a pas de prédécesseurs.
2. Le code suivant convient :

```
s4.predecesseurs = [(s1,1), (s2,2)]
s5.predecesseurs = [(s1,3), (s3,3), (s4,6)]
```

3. `s2.successeurs[1]` renvoie le tuple `(s3,5)` et donc `s2.successeurs[1][1]` renvoie `5` (le nombre de références de **site2** vers **site3**).
4. La popularité de **site1** est $4 + 2 = 6$.
5. Les deux fonctions suivantes conviennent :

```
def calculPopularite(self) :
    self.popularite = 0
    for pred in self.predecesseurs:
        self.popularite += pred[1]
    return self.popularite
```

```
def calculPopularite(self) :
    self.popularite = 0
    for site, ref in self.predecesseurs:
        self.popularite += ref
    return self.popularite
```

6. L'ajout d'un élément se fait à la fin de la liste et la suppression d'un élément se fait au début de la liste. On a donc affaire à une structure de type FIFO, c'est-à-dire une file.
7. Il s'agit d'un parcours en largeur.
8. On obtient la liste `[s1,s3,s4,s5]` contenant `s1` et ses successeurs (et les successeurs des successeurs, mais ce sont les mêmes).
9. Le code suivant convient :

```
def lePlusPopulaire(listeSites):
    maxPopularite = 0
    siteLePlusPopulaire=listeSites[0]
    for site in listeSites:
        if site.popularite > maxPopularite:
            maxPopularite = site.popularite
            siteLePlusPopulaire = site
    return siteLePlusPopulaire
```

10. Parmi les sites **site1**, **site3**, **site4** et **site5** obtenus à la question 8, c'est **site3** qui est le plus populaire. Le code renvoie donc `'site 3'`.
11. La fonction `lePlusPopulaire` est linéaire en le nombre de sommets du graphe elle est donc utilisable sur plusieurs milliers de sites. Cependant le calcul de la popularité demande de parcourir tout les sommets et pour chaque sommet tout ses voisins. Dans le cas d'un graphe complet ceci correspond à une complexité en $O(n^2)$ avec n le nombre de sommets du graphe, ce qui est lent mais encore utilisable.

Exercice 2 (Protocole de routage et bases de données relationnelles)

Partie A

1. On peut, par exemple, citer : résilience de la base des données (persistance des données et intégrité), gestion des accès concurrents, efficacité de traitement des requêtes, sécurisation des accès, etc.
2. On a la route suivante : Bureau No 1 – B – E – A – Prestataire
3. On a les deux routes suivantes avec un coût de 2,3 :
   - Bureau No 2 – C – I – G – F – D – A – Prestataire
   - Bureau No 2 – C – I – H – F – D – A – Prestataire

Partie B

4. Une clé primaire référence de façon unique un enregistrement dans une relation. Comme les autres attributs de la relation `clients` ne peuvent pas être des clés primaires (noms identiques, dates de naissance identiques ou pays identiques pour plusieurs clients), ce ne peut être que `id_client` qui est la clé primaire.
5. Une clé étrangère d'une relation est un attribut (ou un groupe d'attributs) dont les valeurs référencent celles d'une clé primaire d'une autre relation. Ceci permet de relier (jointure) des tables.
   - La relation `croisieres` possède quatre clés étrangères (les quatre escales) qui, chacune, référencent la clé primaire `nom` de la relation `villes`.
   - La relation `villes` n'a pas de clé étrangère.
   - La relation `reservations` admet pour clé étrangère `id_client` qui référence la clé primaire de même nom de la relation `clients`, et `nom_croisiere` qui référence la clé primaire `nom` de la relation `croisieres`.
6. Les clés étrangères de la relation `croisieres` référencent la clé primaire `nom` de la relation `villes`. Les valeurs données à la clé étrangère doivent préexister dans la clé primaire, ce qui ne semble pas être le cas avec la réponse donnée renvoyée par le SGBD. Il aurait déjà fallu ajouter ces noms dans la table `villes`.

Partie C

7. - La première requête permet de récupérer l'identifiant du client à l'aide de toutes les informations connues sur lui.
   - La seconde requête renvoie les identifiants des différentes réservations du client.
8. La requête suivante convient :

```
SELECT id_reservation FROM reservations
JOIN clients ON reservations.id_client = clients.id_client
WHERE clients.nom = 'Barc' AND clients.prenom = 'Jean'
AND clients.date_naissance = '1972/06/29' AND clients.pays = 'Allemagne'
```

9. La requête suivante convient :

```
UPDATE reservations
SET nom_croisiere = 'Croisiere Puerto'
WHERE reservations.id_reservation = 20456
```

10. La requête suivante convient :

```
SELECT clients.nom, clients.prenoms, clients.date_naissance
FROM clients
JOIN reservations ON clients.id_client = reservations.id_client
WHERE reservations.nom_croisiere = 'Croisière Piano'
OR reservations.nom_croisiere = 'Croisière Puerto'
```

Exercice 3 (POO, arbres binaires de recherche et récursivité)

Partie A : la classe `Chien`

1. Il faut saisir l'instruction suivante : `chien40 = Chien(40, 'Duke', 'wheel dog', 10)`
2. La méthode suivante convient :

```
def changer_role(self, nouveau_role):
    self.role = nouveau_role
```

3. Il faut saisir l'instruction suivante : `chien40.changer_role('leader')`

Partie B : la classe `Equipe`

4. Les deux méthodes suivantes conviennent :

```
def retirer_chien(self, numero):
    L = []
    for chien in self.liste_chiens:
        if chien.id_chien != numero:
            L.append(chien)
    self.liste_chiens = L
```

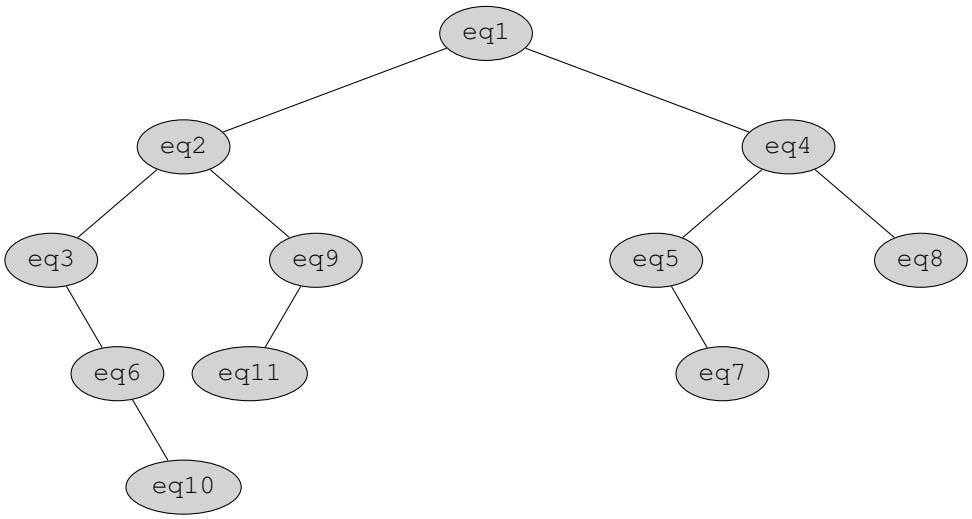
```
def retirer_chien(self, numero) :
    for ind, chien in enumerate(self.liste_chiens) :
        if chien.id_chien == numero :
            self.liste_chiens.pop(ind)
```

5. Il faut saisir l'instruction suivante : `eq11.retirer_chien(46)`
6. On obtient `4.6` (qui est un résultat à interpréter en heure)
7. La fonction suivante convient :

```
def temps_course(equipe):
    cumul = 0
    for temps in equipe.liste_temps:
        cumul += convert(temps)
    return cumul
```

Partie C : classement à l'issue d'une étape

8. On a l'arbre suivant :



9. Comme on a un arbre binaire de recherche, il faut faire un parcours en profondeur infixe.
10. La fonction `inserer` est récursive car elle s'appelle elle-même.
11. On a le code suivant :

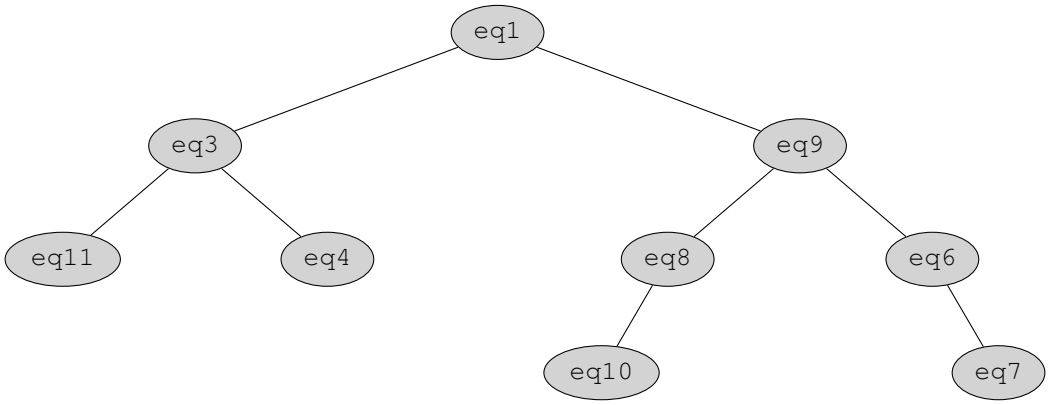
```
def inserer(arb, eq):
    """ Insertion d'une équipe à sa place dans un ABR contenant
        au moins un noeud."""
    if convert(eq.temps_etape) < convert(arb.racine.temps_etape):
        if arb.gauche is None:
            arb.gauche = Noeud(eq)
        else:
            inserer(arb.gauche, eq)
    else:
        if arb.droit is None:
            arb.droit = Noeud(eq)
        else:
            inserer(arb.droit, eq)
```

12. Le code suivant convient :

```
def est_gagnante(arbre):
    if arbre.gauche == None:
        return arbre.racine.nom_equipe
    else:
        return est_gagnante(arbre.gauche)
```

Partie D : classement général

13. L'arbre suivant convient :



14. La fonction suivante convient :

```
def rechercher(arbre, equipe):
    if arbre == None:
        return False
    if arbre.racine is equipe:
        return True
    if temps_course(equipe) < temps_course(arbre.racine):
        return rechercher(arbre.gauche, equipe)
    return rechercher(arbre.droit, equipe)
```

Remarque : l'opérateur == n'est pas définie pour les équipes. L'opérateur `is` utilisé ici est un opérateur d'identité utilisé pour vérifier si deux variables pointent vers le même objet en mémoire ou pour comparer l'identité de deux objets. Il renvoie `True` si les opérandes sont identiques (se réfèrent au même objet).