

# Proposition de correction

## Exercice 1

### Q1

Aucun autre site ne fait référence à Site2

### Q2

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

### Q3

donne le nombre de citations vers le 2ème site : 5

### Q4

$4 + 2 = 6$

### Q5

```
def calculPopularite(self) -> int:  
    popularite = 0  
    for _, liens in self.predecesseurs:  
        popularite += liens  
    return popularite
```

### Q6

file

### Q7

parcours en largeur

### Q8

La liste des sites qui ont été référencés par s1 (incluant s1) : [s1, s3, s4, s5]

### Q9

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

```
siteLePlusPopulaire = site  
return siteLePlusPopulaire
```

### Q10

le nom du site le plus populaire : site1

### Q11

Les listes en Python ne sont pas implémentées comme des listes chaînées mais comme des tableaux dynamiques : la suppression d'un élément au début de la liste avec `pop(0)` implique un décalage de tous les éléments suivants, ce qui a une complexité  $O(n)$ . On est donc en complexité quadratique sans compter du parcours des successeurs.

Il faudrait utiliser une « vraie » implémentation de liste (deque) et un dictionnaire pour stocker les infos sur les nœuds.

## Exercice 2

### Partie A

#### Q1

- intégrité des données
- optimisation des requêtes

#### Q2

bureau 1 → B → A → E → prestataire

#### Q3

- bureau 2 → C → I → G → F → D → A → prestataire, coût =  $0,1 + 1 + 0,1 + 0,1 + 1 = 2,3$
- bureau 2 → C → I → H → F → D → A → prestataire, coût =  $0,1 + 1 + 0,1 + 0,1 + 1 = 2,3$

### Partie B

#### Q4

Permet d'identifier de façon unique les enregistrements de la table clients.

#### Q5

Champ d'une table qui fait référence à la clé primaire d'une autre table afin de mettre les deux tables en relation.

#### Q6

Les valeurs 'Puerto sebo', 'Puerto kifecho', 'Puerto kifebo' et 'Puerto repo' n'existent pas dans la colonne de clé primaire de la table des villes.

Solution : vérifier que les noms des villes dans la requête INSERT correspondent exactement aux noms des villes dans la table des villes.

## Partie C

### Q7

- La première requête SQL est utilisée pour rechercher l'identifiant (id) du client.
- La deuxième requête SQL est utilisée pour rechercher les identifiants des réservations effectuées par le client.

### Q8

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

### Q9

```
UPDATE reservations
SET nom_croisiere = 'Croisière Puerto'
WHERE id_reservation = 20456
```

### Q10

```
SELECT clients.nom, clients.prenom, clients.date_naissance
FROM clients
JOIN reservations ON clients.id = reservations.id_client
WHERE reservations.nom IN ('Croisière Piano', 'Croisière Puerto')
ORDER BY clients.nom, clients.prenom
```

## Exercice 3

### Partie A

#### Q1

```
chien40 = Chien(40, 'Duke', 'wheel dog', 10)
```

#### Q2

```
def changer_role(self, nouveau_role : str):
    """Change le rôle du chien avec la valeur passée en paramètre."""
    self.role = nouveau_role
```

#### Q3

```
chien40.changer_role('leader')
```

## Partie B

## Q4

```
def retirer_chien(self, numero : int) -> bool:
    """
    @param numero -- un entier correspondant au numéro attribué au chien
    lors de l'inscription
    @remark mettre à jour l'attribut liste_chiens après retrait du chien
    dont la valeur de l'attribut id_chien est numero.
    @return True si chien trouvé, False sinon (évite de parcourir toute la liste)
    """
    if type(numero) is int:
        for chien in self.liste_chiens:
            if chien.id_chien == numero:
                self.liste_chiens.remove(chien)
                return True
        return False
```

## Q5

```
eq11.retirer_chien(chien46.id_chien)
```

## Q6

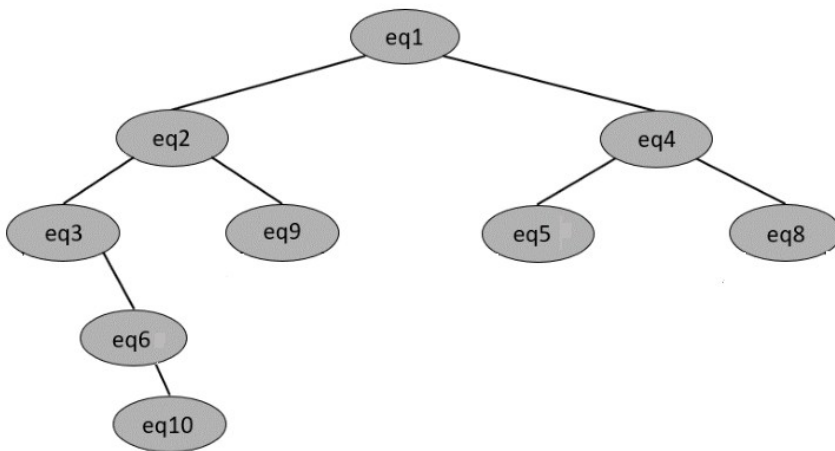
4.6

## Q7

```
def temps_course(equipe : Equipe) -> float:
    """
    @param equipe -- une Equipe
    @return cumul des temps de l'équipe equipe à l'issue des 9 étapes de la course
    """
    cumul = 0.
    for temps in equipe.liste_temps:
        cumul += convert(temps)
    return cumul
```

Partie C

Q8



Q9

parcours infixe

Q10

la fonction s'appelle elle-même

Q11

```

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)
  
```

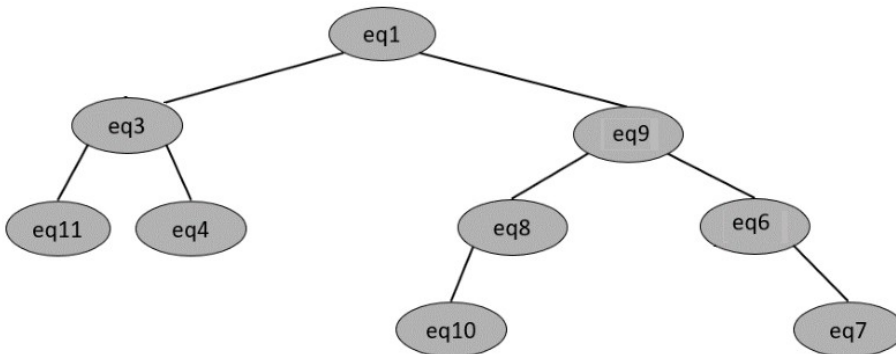
Q12

```

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

## Partie D

## Q13



## Q14

```
def rechercher(arbre, equipe):  
    """  
    Paramètres  
    -----  
    arbre : un ABR, non vide, de type Noeud, représentant le classement général.  
    equipe : un élément, de type Equipe  
             dont on veut déterminer l'appartenance ou non à l'ABR arbre.  
    Résultat  
    -----  
    Cette fonction renvoie True si equipe est un nœud de arbre, False sinon.  
    """  
    if arbre is None:  
        return False  
  
    if temps_course(equipe) < temps_course(arbre.racine):  
        return rechercher(arbre.gauche, equipe)  
    elif temps_course(equipe) > temps_course(arbre.racine):  
        return rechercher(arbre.droit, equipe)  
    else:  
        if arbre.racine is equipe:  
            return True  
        else:  
            # en cas de temps identiques  
            return rechercher(arbre.droit, equipe) or rechercher(arbre.gauche, equipe)
```