25NSIJ2G11

Exercice 1 (6 points)

Cet exercice porte sur les bases de données relationnelles et les requêtes SQL.

1. Le rôle des clés primaires est de pouvoir identifier de façon unique un enregistrement d'une table.
2. Sans le champ `id_match`, on ne pourrait pas stocker dans la table `match` deux matchs entre les mêmes équipes dans le même ordre et avec le même score final (et donc le même gagnant).
3. Sur l'extrait présenté, `SELECT prenom FROM joueur WHERE ann_naiss < 1985;` renvoie la table :

prenom
Henri
Laure
Brigitte
Laure

4. `SELECT DISTINCT prenom FROM joueur WHERE ann_naiss < 1985;` permet d'éviter les doublons.
5. `SELECT nom, ann_naiss, num_port FROM joueur WHERE commune="Bois-Plage";` convient.
6. On utilise une jointure puisqu'on a besoin des deux tables `joueur` et `equipe`.
   ```
   SELECT joueur.nom, prenom FROM joueur JOIN equipe ON joueur.id_joueur = equipe.j_1
   WHERE equipe.nom = "Les Kangourous";
   ```
7. La requête `UPDATE equipe SET points = 5 WHERE nom = "Volley Warriors";` effectue la mise à jour demandée.
8. La requête `DELETE FROM joueur WHERE id_joueur = 35;` supprime le joueur d'identifiant 35 (en supposant que ce joueur est d'abord supprimé de l'équipe des Volley Warriors dans la table `equipe` faute de quoi on peut avoir une erreur de violation de contrainte de référence).
9. `SELECT id_match FROM match WHERE eq_1 = 12 OR eq_2 = 12;` convient.
10. `SELECT id_match FROM match JOIN equipe ON match.eq_1 = equipe.id_equipe JOIN joueur ON equipe.j_1 = joueur.id_joueur WHERE commune = "Bois-Plage";` convient.
11. `SELECT DISTINCT joueur.nom, prenom FROM joueur JOIN equipe ON joueur.id_joueur = equipe.j_1 JOIN match ON equipe.id_equipe = eq_gagnante ORDER BY joueur.nom, prenom ;`
    permet de joindre les trois tables pour répondre à la question posée.

Exercice 2 (6 points)

Cet exercice porte sur les listes, les dictionnaires, les fonctions et la récursivité

Partie A

1. En ASCII, `'E'` est codé par `0x45` et `'W'` par `0x57`, la somme est `0x9C` soit une clé égale à $9 \times 16 + 12 = 156$.
2. Les mots `'SAC'` et `'CAS'` ont la même clé puisqu'ils contiennent exactement les mêmes lettres, on ajoute les même codes ASCII.
3. On complète les lignes 2 à 4 dans la fonction `code_hachage`.

```
1  def code_hachage(mot):
2      somme = 0
3      for caractere in mot:
4          somme = somme + ord(caractere)
5      return somme % 0x100
```

4. L'expression `somme % 0x100` permet d'obtenir le reste dans la division entière de somme par `0x100` c'est-à-dire 256. Autrement dit on ne garde que l'octet de poids faible de la somme, comme voulu.

Partie B

5. La boucle `while` des lignes 3-10 contient exactement une comparaison de chaînes ligne 4. Or cette boucle est exécutée au pire des cas `len(liste)` comme on peut le voir grâce au variant de boucle `i`. La complexité est donc linéaire en n.
6. L'expression booléenne `c in dico` permet de savoir si la clé `c` est une clé du dictionnaire `dico`.
7. On écrit une fonction `ajouter_mot_dict`.

```
def ajouter_mot_dict(dict_mots, mot):
    h = code_hachage(mot)
    if h in dict_mots:
        ajouter_mot_liste(dict_mots[h], mot)
    else:
        dict_mots[h]=[mot]
```

Partie C

8. Dans l'exemple donné, les valeurs successives de `debut` et `fin` sont données dans le tableau ci-dessous.

appel	0	1	2
`debut`	0	0	1
`fin`	5	1	1

9. Une recherche simple nécessite dans le pire des cas d'examine r tous les mots un par un un, alors qu'avec une recherche dichotomique on élimine la moitié des mots candidats à chaque comparaison.
10. La complexité de la recherche dichotomique est logarithmique, en $O(\log_2(n))$ pour une liste de longueur n.
11. On écrit la fonction demandée.

```
def mot_present(dict_mots, mot):
    h = code_hachage(mot)
    mots = dict_mots[h]
    return est_present(mots, mot, 0, len(mots))
```

Exercice 3 (8 points)

Cet exercice porte sur les graphes, la programmation objet et la récursivité.

1. Le chemin 3 -> 9 -> 1 -> 2 -> 4 -> 8 est non prolongeable.

Partie A

2. Le `for valeur in range(1, n+1)` signifie que la variable `valeur` prendra les valeurs `1, 2 , ..., n` bornes incluses.
3. Après l'exécution du code `jeu_9 = creer_jeu(9)`, la valeur de `jeu_9[0].valeur` est `1`.
4. À la ligne 5 on vérifie que la valeur de `s` est un diviseur de la valeur de `self` et que `s` et `self` sont distincts.
5. Après l'exécution de `jeu_9[5].relier_diviseurs(jeu)`, le contenu de `jeu_9[5].diviseurs` la liste des sommets dont la valeur divise 6, `[jeu_9[1], jeu_9[1], jeu_9[2]]`.
6. On écrit la ligne 6.

```
6           sommet.relier_diviseurs(jeu)
```

7. On complète la ligne 3 du code de la méthode `liste_diviseurs`.

```
3           return [sommet.valeur for sommet in self.diviseurs]
```

8. On complète le jeu de tests en utilisant l'exemple `jeu9`.

```
l_div_3 = jeu9[2].lister_diviseurs()
l_mult_3 = jeu9[2].lister_multiples()
assert 1 in l_div_3
assert 6 in l_mult_3
assert 9 in l_mult_3
```

Partie B

9. La ligne `assert not self.est_vide()` interrompra l'exécution du programme si la file est vide.
10. On complète le code de la méthode `taille`.

```
    def taille(self):
        return len(self.donnees) - self.decalage
```

11. On écrit le jeu de test demandé.

```
f = File()
f.enfiler(1)
f.enfiler(2)
f.enfiler(3)
assert f.taille() == 3
u = f.defiler()
assert u == 1
v = f.defiler()
assert v == 2
w = f.defiler()
assert w == 3
assert f.est_vide()
```

Partie C

12. On complète le code de la fonction `rechercher_chemins`.

```
def rechercher_chemins(jeu):
    chemins_np = []
    f = File()
    for sommet in jeu:
        f.enfiler([sommet])
        while not f.est_vide():
            chemin = f.defiler()
            dernier = chemin[-1]
            voisins = dernier.diviseurs + dernier.multiples
            prolongeable = False
            for voisin in voisins:
                if voisin not in chemin:
                    prolongeable = True
                    f.enfiler(chemin + [voisin])
            if not prolongeable:
                chemins_np.append(chemin)
    return chemins_np
```

13. On écrit une fonction `valeurs_chemins`.

```
def valeurs_chemin(chemin):
    return [sommet.valeur for sommet in chemin]
```

14. On complète les lignes demandées.

```
chemins = []
for chemin in rechercher_chemins(jeu9):
    chemins.append(valeurs_chemin(chemin))
```

15. On écrit le code de la fonction `extraire_plus_longs_chemins`.

```
def extraire_plus_longs_chemins(L_chemins):
    plc=[]  # plus longs chemins
    lmax=0  # longueur maximale
    for chemin in L_chemins:
        if len(chemin) > lmax:
            plc = [chemin]
        elif len(chemin) == lmax:
            pls.append(chemin)
    return plc
```

16. Le nombre de chemins semble croître de manière rapide, peut-être exponentielle. Cela signifie que pour un nombre de sommets de l'ordre de quelques dizaines on aura besoin d'un meilleur algorithme, ou bien d'un plus gros ordinateur.