Spécialité Numérique et Sciences Informatiques
SUJET J2 -2025

Correction

Exercice 1

Q1 Les clés primaires permettent d'identifier de manière unique chaque enregistrement dans une table.

Q2 Sans le champ `id_match`, il serait difficile de référencer de manière unique chaque match, surtout lorsque plusieurs matchs peuvent impliquer les mêmes équipes avec le même score (une équipe peut jouer plusieurs matchs). Cela entraînerait des problèmes pour distinguer les différents matchs.

Q3 Le résultat de la requête est :

prenom
Henri
Laure
Brigitte
Laure

Q4 Pour éviter les doublons, on peut utiliser le mot-clé DISTINCT :

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

Q5

```
SELECT nom, ann_naiss, num_port FROM joueur WHERE commune = 'Bois-Plage'
```

Q6

```
SELECT joueur.nom, joueur.prenom
FROM joueur
JOIN equipe ON joueur.id_joueur = equipe.j_1
WHERE equipe.nom = 'Les Kangourous'
```

Q7

```
UPDATE equipe SET points = 5 WHERE nom = 'Volley Warriors'
```

Q8 Avant de supprimer le joueur avec l'identifiant 35, il faut vérifier qu'il n'est référencé dans aucune équipe. Or c'est le cas pour l'équipe *Volley Warriors*, la suppression n'est pas autorisée sans d'abord mettre à jour ou supprimer les références, d'où les instructions suivantes :

```
UPDATE equipe SET j_4 = NULL WHERE j_4 = 35;
DELETE FROM joueur WHERE id_joueur = 35
```

Q9

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

Q10

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

Q11

```sql
SELECT DISTINCT joueur.nom, joueur.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, joueur.prenom
```

Exercice 2

Partie A

Q1 La valeur hexadécimale de la clé associée au mot 'EW' est $0x9C$.

Q2 La clé associée au mot 'SAC' sera la même que celle associée au mot 'CAS', car les deux mots ont les mêmes lettres. Ainsi, ils auront la même somme de codes ASCII.

Q3 Voici le code complété pour la fonction `code_hachage` :

```
def code_hachage(mot):
    somme = 0
    for caractere in mot :
        somme = somme + ord(caractere)
    return somme % 0x100
```

Q4 L'expression `somme % 0x100` permet de conserver uniquement l'octet de poids faible de la somme. Cela signifie que la clé sera toujours un nombre entre 0 et 255, puisque l'on fait un modulo $0x100$ soit 256, ce qui correspond à toutes les valeurs possibles d'un octet (8 bits).

Partie B

Q5 Dans le pire cas, le nombre de comparaisons sera de l'ordre de n, où n est le nombre de mots présents dans la liste,chaque boucle `while` faisant une comparaison.

Q6 Pour tester si une clé `c` est présente dans un dictionnaire `dico`, on utilise l'expression `c in dico`.

Q7 Voici un code possible pour la fonction `ajouter_mot_dict` :

```
def ajouter_mot_dict(dict_mots, mot):
    cle = code_hachage(mot)
    if cle in dict_mots:
```

```
                dict_mots[cle] = ajouter_mot_liste(dict_mots[cle], mot)
            else:
                dict_mots[cle] = [mot]
            return dict_mots
```

Partie C

Q8 Les valeurs des paramètres `debut` et `fin` lors de chaque appel de la fonction `est_present` pour l'exemple donné sont les suivantes :

- Premier appel : debut = 0, fin = 5
- Deuxième appel : debut = 0, fin = 2 (puisque le milieu est 2 et on cherche avant)
- Troisième appel : On trouve 'NSI', debut = 1 et fin =1.

Q9 La méthode dichotomique permet d'effectuer moins d'opérations car elle divise à chaque étape la taille de la liste à explorer par deux, contrairement à une recherche simple qui explore chaque élément un par un.

Q10 L'ordre de grandeur du nombre de comparaisons de mots effectuées par la méthode dichotomique est de $\mathcal{O}(\log n)$, où n est la longueur de la liste.

Q11 Voici un code possible pour la fonction `mot_present` :

```
        def mot_present(dict_mots, mot):
            cle = code_hachage(mot)
            if cle in dict_mots:
                liste = dict_mots[cle]
                return est_present(liste, mot, 0, len(liste))
            else:
                return False
```

Exercice 3

Partie A

Q1 Un chemin non prolongeable qui commence par $3 \to 9 \to 1 \to 2$ pourrait être : $3 \to 9 \to 1 \to 2 \to 4 \to 8$.

Q2 Les valeurs prises par la variable `valeur` lors de l'exécution de `creer_jeu(9)` sont de 1 à 9 inclus.

Q3 La valeur de `jeu_9[0].valeur` après l'exécution de `creer_jeu(9)` est 1.

Q4 Le test `s != self` vérifie que l'on ne va pas diviser le sommet par lui-même et le test `self.valeur % s.valeur == 0` vérifie si la valeur du sommet actuel est divisible par la valeur du sommet `s`, ce qui signifie que `s` est un diviseur du sommet actuel.

Q5 Après l'exécution de `jeu_9[5].relier_diviseurs(jeu)`, `jeu_9[5].diviseurs` contiendra les sommets correspondant aux diviseurs de 6, qui sont 1, 2, et 3 soit `[1, 2, 3]`

Q6 La ligne manquante est :

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

Q7 Voici le code complété pour la méthode `lister_diviseurs` :

```
    def lister_diviseurs(self):
        """Renvoie la liste des diviseurs de ce sommet"""
        return [sommet.valeur for sommet in self.diviseurs ]
```

Q8 Voici un jeu de tests complété :

```
l_div_3 = jeu_9[2].lister_diviseurs()
l_mult_3 = jeu_9[2].lister_multiples()
assert 1 in l_div_3
assert 6 in l_mult_3
assert 9 in l_mult_3
```

Partie B

Q9 La ligne de code `assert not self.est_vide()` vérifie que la file n'est pas vide avant de tenter de retirer un élément. Si la file est vide, cela produit une assertion error.

Q10 Voici le code complété pour la méthode `taille` :

```
    def taille(self):
        """Renvoie le nombre d'éléments présents dans la file"""
        return len(self.donnees) - self.decalage
```

Q11 Voici une séquence d'instructions pour tester la classe `File` :

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

Partie C

Q12 Voici le code complété pour 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
```

Q13 Voici une fonction pour obtenir les valeurs d'un chemin :

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

Q14 Voici le code complété pour obtenir tous les chemins non prolongeables du jeu à 9 entiers :

```
                chemins = []
                for chemin in rechercher_chemins(jeu_9):
                    chemins.append(valeurs_chemin(chemin))
```

Q15 Voici la fonction pour extraire les plus longs chemins :

```
def extraire_plus_longs_chemins(L_chemins):
        if not L_chemins:
  return []
        long_max = 0
        for chemin in L_chemins:
                if len(chemin) > long_max:
                        long_max = len(chemin)
        return [chemin for chemin in L_chemins if len(chemin) == long_max]
```

Q16 Non, on ne peut pas résoudre le jeu pour un nombre de sommets aussi grand que l'on souhaite. En effet, le nombre de chemins non prolongeables augmente de manière exponentielle avec le nombre de sommets, comme montré dans le tableau. Pour un grand nombre de sommets, le temps de calcul nécessaire deviendra prohibitif.