25NSIJ2JA1

Exercice 1 (6 points)

Cet exercice porte sur la programmation orientée objet et l'algorithmique.

Partie A

1. On complète le constructeur.

```
3           self.jour = jour
4           self.mois = mois
5           self.annee = annee
```

2. La date qui correspond à `d = Date(1, 5, 2000)` est le 1er mai 2000.
3. On instancie la date demandée par `d = Date(19, 6, 2024)`.
4. On complète la méthode `get_annee`.

```
15      def get_annee(self):
16          return self.annee
```

5. On complète la méthode `set_mois`.

```
21      def set_mois(self, mois):
22          self.mois = mois
```

6. On insère les lignes suivantes dans le constructeur.

```
7       if self.est_bissextile():
8           self.nb_jours_par_mois[1] = 29
```

Partie B

7. On écrit le code de la méthode `est_bissextile`.

```
        def est_bissextile(self):
            a = self.annee
            return a%4 ==0 and a%100!=0 or a%400 == 0
```

8. L'affichage sera 79 (résultat du calcul $20 + 31 + 28$).
9. On complète la méthode `nb_jours_restants`.

```
        def nb_jours_restants(self):
            j = 365
            if self.est_bissextile():
                j = 366
            return j - self.nb_jours_passes()
```

Partie C

10. On présente les affichages obtenus en console.

```
>>> d1.nb_jours_depuis(d2)
0
>>> d1.nb_jours_depuis(d3)
-1
>>> d1.nb_jours_depuis(d4)
-1
>>> d1.nb_jours_depuis(d5)
731
```

11. On complète la méthode `timestamp`.

```
    def timestamp(self):
        d = Date(1, 1, 1970) # epoch
        return self.nb_jours_depuis(d) * 24 * 3600
```

Exercice 2 (6 points)

Cet exercice porte sur la programmation Python, la gestion des processus.

1. Ce programme est un ordonnanceur.
2. Les états possibles d'un processus sont élu, prêt, bloqué.
3. Proposition 2, comme indiqué explicitement par l'énoncé.
4. On complète le constructeur.

```
class Processus:
    __init__(self, PID, priorite, temps_CPU):
        self.priorite = priorite
        self.PID = PID
        self.temps_utilisation = 0
        self.temps_CPU = temps_CPU
```

5. On complète la simulation, en supposant que les files sont implémentées par des `list` avec enfilement par `f.insert(0)` et défilement par `f.pop()`.

```
Cycle 1 :  CPU=P1  liste_files=[[P3, P2], [], []]
Cycle 2 :  CPU=P2  liste_files=[[P3],[P1],[]]
Cycle 3 :  CPU=P3  liste_files=[[], [P2, P1],[]]
Cycle 4 :  CPU=P3  liste_files=[[], [P2, P1], []]
Cycle 5 :  CPU=P1  liste_files=[[], [P2], [P3]]
```

6. Chaque nouveau processus court arrive avec la priorité maximale zéro, tandis que le processus long a une priorité qui s'éloigne de zéro à chaque fois qu'il s'exécute ; en particulier au bout de quatre cycles le processus long ne sera plus jamais prioritaire sur les processus courts, qui sont sans cesse renouvelés, donc le processus long ne sera plus jamais élu.
7. Au bout de `Max_Temps` cycles en attente, la priorité du processus long augmente, donc en temps fini ce processus sera élu, et puisqu'il a besoin d'un temps CPU fini il termine.
8. On écrit la fonction `meilleur_priorite`.

```
def meilleur_priorite(liste_files):
    m=0
    while m < len(liste_files) and liste_files[m] == []:
        m = m + 1
    if m < len(liste_files):
        return m
    return None
```

9. On écrit la fonction `prioritaire`.

```
def prioritaire(liste_files):
    m = meilleur_priorite(liste_files)
    if m is not None:
        return liste_files[m].pop()
    return None
```

10. On écrit la fonction `gerer`. Bien que l'énoncé soit peu explicite sur ce point, on suppose que le paramètre `p` peut être `None`, faute de quoi il se peut qu'aucun processus ne soit élu malgré la présence de processus prêts. On interprétera l'énoncé par « la fonction `gerer` met à jour les structures et renvoie le nouveau processus élu ».

```python
def gerer(p, liste_files):
    if p is None:
        return prioritaire(liste_files)
    if p.temps_utilisation + 1 == p.temps_CPU:
        return None
    else:
        p.temps_utilisation += 1
            m = meilleure_priorite(liste_files)
            if m is not None and m <= p.priorite:
                p.priorite +=1
                if p.priorite < len(liste_files):
                    liste_files[p.priorite].insert(0,p)
                else:
                    liste_files.append([p])
                return prioritaire(liste_files)
            if m is None:
                p.priorite +=1
                return p
```

On peut aussi imaginer que la fonction `gerer` ne se contente pas de renvoyer le prochain processus à exécuter, mais gère tout l'ordonnancement jusqu'à ce que tous les processus aient terminé.

```python
def gerer(p, liste_files):
    """
    On suppose qu'initialement on appelle cette fonciont avec p = None
    """
    if p is None:
        p = prioritaire(liste_files)
        if p is None:
            print("fin")
        else: # p est alors en cours d'exécution
            #print(p, liste_files)
            gerer(p, liste_files)
    else: # p est alors en cours d'exécution
        if p.temps_utilisation + 1 == p.temps_CPU:
            gerer(None, liste_files)
        else:
            p.temps_utilisation += 1
            m = meilleur_priorite(liste_files)
            if m is None:
                #print(p, liste_files)
                gerer(p, liste_files)
            else:# il reste encore des processus dans les files d'attente
                if m <= p.priorite:
                    p.priorite += 1
                    if m + 1 < len(liste_files):
                        liste_files[m + 1].insert(0, p)
                    else:
                        liste_files.append([p])
                    p = prioritaire(liste_files)
                    #print(p, liste_files)
                    gerer(p, liste_files)
                else:
                    p.priorite += 1
                    #print(p, liste_files)
                    gerer(p, liste_files)
```

On peut décommenter les `print` pour avoir une trace de l'exécution et vérifier sa réponse à la question 5.

Exercice 3 (8 points)

Cet exercice porte sur la programmation Python (dictionnaire, récursivité, spécification), la programmation orientée objet, les bases de données relationnelles, les requêtes SQL et les arbres binaires.

Partie A

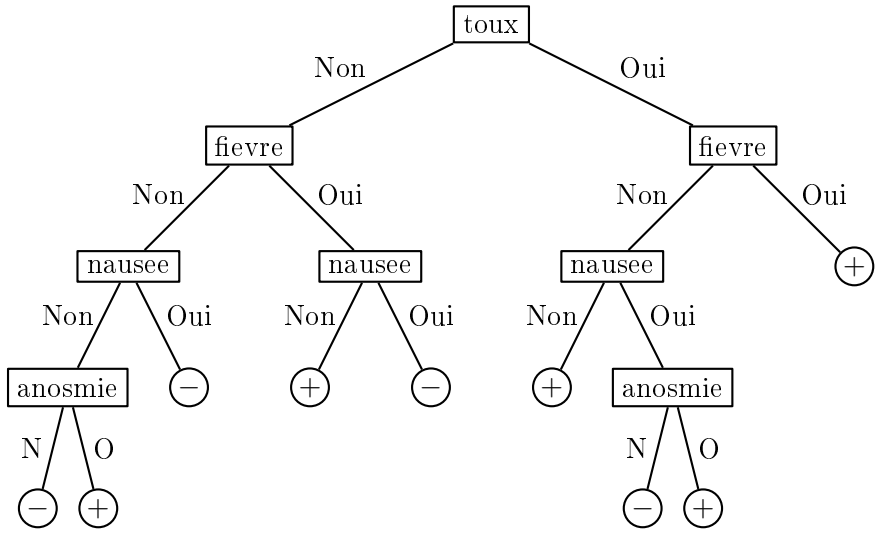
1. La requête `SELECT nom_patient, prenom FROM Patient WHERE age > 60;` convient.
2. La requête `UPDATE Symptome SET toux='Non' WHERE nom_patient='Heartman';` convient.
3. La requête `SELECT COUNT(*) FROM Diagnostic JOIN Symptome ON Diagnostic.nom_patient=Symptome.nom_patient WHERE nom_maladie="Covid-19" AND toux='Oui';` convient.
4. La requête provoque une erreur car la clé primaire `Douglas` est déjà utilisée.
5. On pourrait utiliser très classiquement un attribut supplémentaire entier `id_patient` dans la relation `Patient` qui servirait de clé primaire.

Partie B

6. Le diagnostic est positif pour le patient considéré.
7. L'assertion de la ligne 14 permet de vérifier que `self` n'est pas une feuille et donc que `self.valeur` est effectivement un symptôme.
8. La classe `Noeud` possède les attributs `valeur`, `gauche` et `droit` et les méthodes `est_feuille`, `symptome` et `diagnostic`.
9. On complète la fonction.

```
1  def applique(arbre, patient):
2      if arbre.est_feuille():
3          return arbre.valeur
4      else:
5          if patient[arbre.symptome()]:
6              return applique(arbre.droit, patient)
7          else:
8              return applique(arbre.gauche, patient)
```

10. La taille de l'arbre est 31.
11. On réduit l'arbre (récursivement).



12. On complète la méthode.

```
    def réduire(self):
        """fonction récursive qui réduit la taille d'un arbre de
        décision sans changer les décisions prises"""
        if self.est_feuille():
            return
        self.gauche.reduire()
        self.droit.reduire()
        if self.gauche.est_feuille() and self.droit.est_feuille() \
            and self.gauche.valeur == self.droit.valeur:
            self.valeur = self.gauche.valeur
            self.gauche = None
            self.droite = None
```

Partie C

13. On complète la fonction.

```
def verifie(num_secu):
    n = num_secu // 100
    k = num_secu % 100
    return (n + k) % 97 == 0
```

14. On complète la fonction.

```
def cle(n):
    return 97 - n % 97
```