Métropole - sujet 2 - juin 2021 (corrigé)

Exercice 1 (SQL)

1. La première et la troisième requête utilisent toutes les deux la même valeur pour l'attribut `idEleves` (128). L'attribut `idEleve` étant une clé primaire, nous allons donc avoir une erreur (on ne doit pas trouver dans toute la relation deux fois la même valeur pour une clé primaire).
2. Dans la relation `Emprunts` l'attribut `idEleve` est une clé étrangère, c'est ce qui assure que l'on ne pourra pas enregistrer un emprunt pour un élève qui n'a pas encore été inscrit dans la relation `Eleves`.
3. La requête suivante convient :

```sql
SELECT titre
FROM Livres
WHERE auteur = 'Molière'
```

4. Cette requête permet d'avoir le nombre d'élèves de la classe T2 inscrits au CDI.
5. La requête suivante convient :

```sql
INSERT INTO Emprunts
VALUES (640, 192, '9782070409228', '2020-09-15', NULL)
```

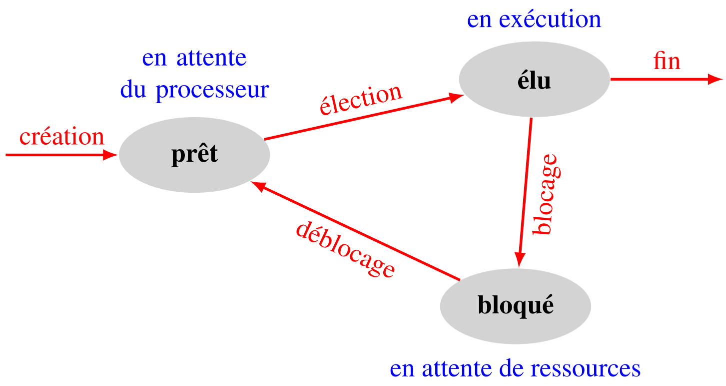
6. Cette requête permet d'avoir le nom et le prénom de tous les élèves de la classe T2 qui ont déjà emprunté un livre au CDI.
7. La requête suivante convient :

```sql
SELECT nom, prenom
FROM Emprunts
INNER JOIN Livres ON Livres.isbn = Emprunts.isbn
INNER JOIN Eleves ON Eleves.idEleve = Emprunts.idEleve
WHERE titre = 'Les misérables'
```

Exercice 2 (Processus)

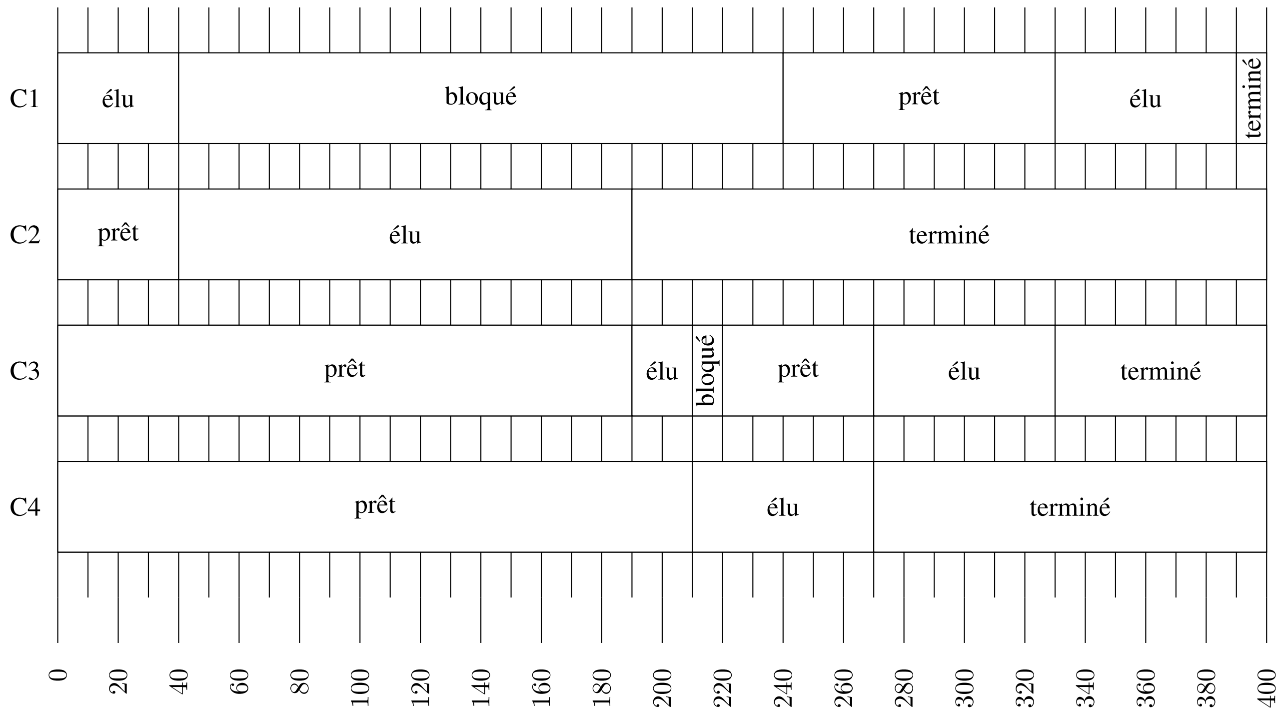
1. (a) Quand un processus est dans l'état élu, cela signifie que ce même processus est en cours d'exécution.

   (b) Le schéma suivant convient :



2. (a) Il s'agit de la réponse ii : premier entré, premier sorti.

   (b) On a le schéma suivant :

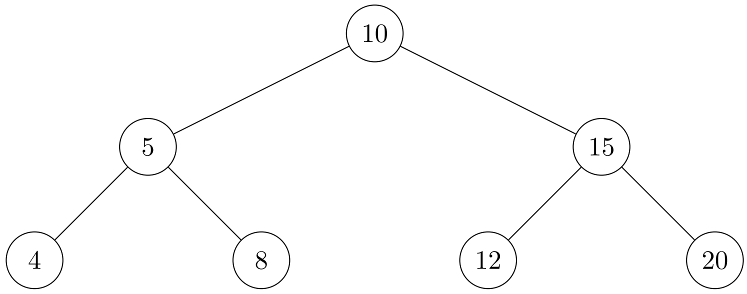


3. (a) P1 verrouille le fichier_1 et P2 verrouille le fichier_2.
   P1 attend le fichier_2 avant de pouvoir effectuer les calculs (et donc libérer le fichier_1).
   Dans le même temps, P2 attend le fichier_1 avant de pouvoir effectuer les calculs (et donc libérer le fichier_2).
   Nous avons donc une situation d'interblocage.

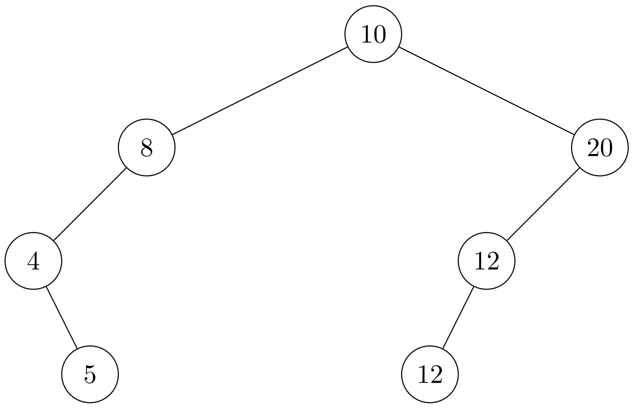
   (b) Il suffit d'inverser les deix premières actions pour le programme 2.

Exercice 3 (Arbre binaire de recherche et POO)

1. (a) La taille de l'arbre est égale au nombre de nœuds, donc à 7.
   (b) La hauteur de l'arbre est 4.
2. L'arbre suivant convient :



3. On obtient l'arbre suivant :



4. La fonction suivante convient :

```python
def hauteur(self):
    if self.gauche == None and self.droit == None:
        return 1
    if self.gauche == None:
        return 1+self.droit.hauteur()
    elif self.droit == None:
        return 1+self.gauche.hauteur()
    else:
        hg = self.gauche.hauteur()
        hd = self.droit.hauteur()
        if hg > hd:
            return hg+1
        else:
            return hd+1
```

5. ⋆ Méthode `taille` de la classe `Noeud` :

```python
def taille(self):
    if self.gauche == None and self.droit == None :
        return 1
    if self.gauche == None :
        return 1 + self.droit.taille()
    elif self.droit == None :
        return 1 + self.gauche.taille()
    else :
        return 1 + self.gauche.taille() + self.droit.taille()
```

⋆ Méthode `taille` de la classe `Arbre` :

```
def taille(self):
    return self.racine.taille()
```

6. (a) L'arbre doit être complet sur les $h-1$ premiers niveaux (donc $2^{h-1}-1$ nœuds) et on ajoute un nœud sur le dernier niveau, donc $t_{\min}=2^{h-1}$.

   (b) Un arbre de hauteur h est bien construit s'il est complet sur les $h-1$ premiers niveaux et s'il admet au moins un nœud sur le dernier niveau, c'est-à-dire si sa taille est supérieure ou égale à $t_{\min}$. La fonction suivante convient donc :

```
def bien_construit(self):
    t = self.taille()
    h = self.hauteur()
    return t >= 2**(h-1)
```

Exercice 4 (Programmation et récursivité)

1. Prenons un exemple où au départ on a : `lst[i1] = 3` et `lst[2] = 8` :
   - ⋆ après la ligne `lst[i2] = lst[i1]`, nous avons `lst[i2] = 3`
   - ⋆ après la ligne `lst[i1] = lst[i2]`, nous avons `lst[i1] = 3`

   Le résultat attendu était `lst[i1] = 8` et `lst[2] = 3` et le résultat obtenu est `lst[i1] = 3` et `lst[2] = 3`. Le code `Python` proposé ne réalise donc pas l'échange attendu. Il faut utiliser une variable temporaire pour que cela fonctionne :

```python
def echange(lst, i1, i2):
    lst[i2] = lst[i1]
    lst[i1] = lst[i2]
```

2. Les valeurs qui pourront être renvoyées par `randint(0, 10)` sont : `0`, `1`, `9` et `10`.
3. (a) Nous avons un appel récursif avec `melange(lst, ind-1)`. A chaque appel récursif on soustrait `1` au paramètre `ind`. Au bout d'un certain nombre d'appels récursifs, le paramètre sera donc égal à `0` et les instructions « contenues » dans le `if` (`if ind>0`) ne seront plus exécutées. Le programme s'arrêtera.

   (b) Pour l'appel initial de la fonction nous avons `lst = n-1`. Pour le premier appel récursif nous avons `lst = n-2`. Pour le dernier appel récursif nous avons `lst = 0`. Nous avons donc eu `n-1` appels récursifs.

   (c) On a l'affichage suivant :

```
[0, 1, 2, 3, 4]
[0, 1, 4, 3, 2]   # j = 2
[0, 3, 4, 1, 2]   # j = 1
[0, 3, 4, 1, 2]   # j = 2
[3, 0, 4, 1, 2]   # j = 0
```

   (d) La fonction suivante convient :

```python
def melange(lst):
    ind = len(lst)-1
    while ind > 0 :
        j = randint(0, ind)
        echange (lst, ind, j)
        ind = ind - 1
```

Exercice 5 (Programmation)

1. (a) Si les éléments du tableau sont tous positifs, il suffit d'additionner tous les éléments du tableau pour obtenir la somme maximale (la sous-séquence correspond à l'ensemble du tableau).

   (b) Si les éléments du tableau sont tous négatifs, il suffit de prendre l'élément le plus grand du tableau (la sous-séquence est réduite à un seul élément).

2. (a) La fonction suivante convient :

```python
def somme_sous_sequence(lst, i, j):
    somme = 0
    for ind in range(i,j+1):
        somme = somme + lst[ind]
    return somme
```

   (b) Pour un tableau de dix éléments, nous avons $10+9+8+7+6+5+4+3+2+1=55$ comparaisons.

   (c) La fonction suivante convient :

```python
def pgsp(lst):
    n = len(lst)
    somme_max = lst[0]
    i_max = 0
    j_max = 0
    for i in range(n):
        for j in range(i,n):
            s = somme_sous_sequence(lst,i,j)
            if s > somme_max:
                somme_max = s
                i_max = i
                j_max = j
    return (somme_max, i_max, j_max)
```

3. (a) On a le tableau suivant :

i	0	1	2	3	4	5	6	7
lst[i]	-8	-4	6	8	-6	10	-4	-4
S(i)	-8	-4	6	14	8	18	14	10

   (b) La fonction suivante convient :

```python
def pgsp2(lst):
    sommes_max = [lst[0]]
    for i in range(1, len(lst)):

        # à compléter

    return max(sommes_max)
```

   (c) Cette solution est plus avantageuse, car la complexité en temps de l'algorithme est en $O(n)$ alors que dans le cas précédent il était en $O(n^2)$.