Éléments de correction sujet 05

Exercice 1

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 2 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

```
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

```
UPDATE Emprunts
SET dateRetour = '2020-09-30'
WHERE idEmprunt = 640
```

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

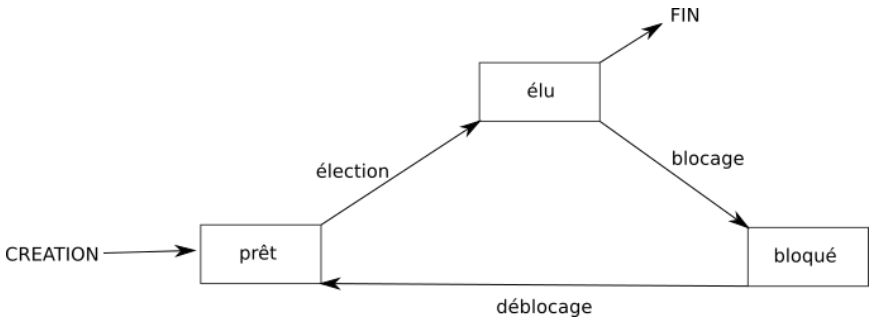
```
SELECT nom, prenom
FROM Emprunts
INNER JOIN Livres ON Livres.isbn = Emprunts.isbn
INNER JOIN Eleves ON Eleves.idEleve = Emprunts.idEleve
WHERE titre = 'Les misérables'
```

Exercice 2

1a

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

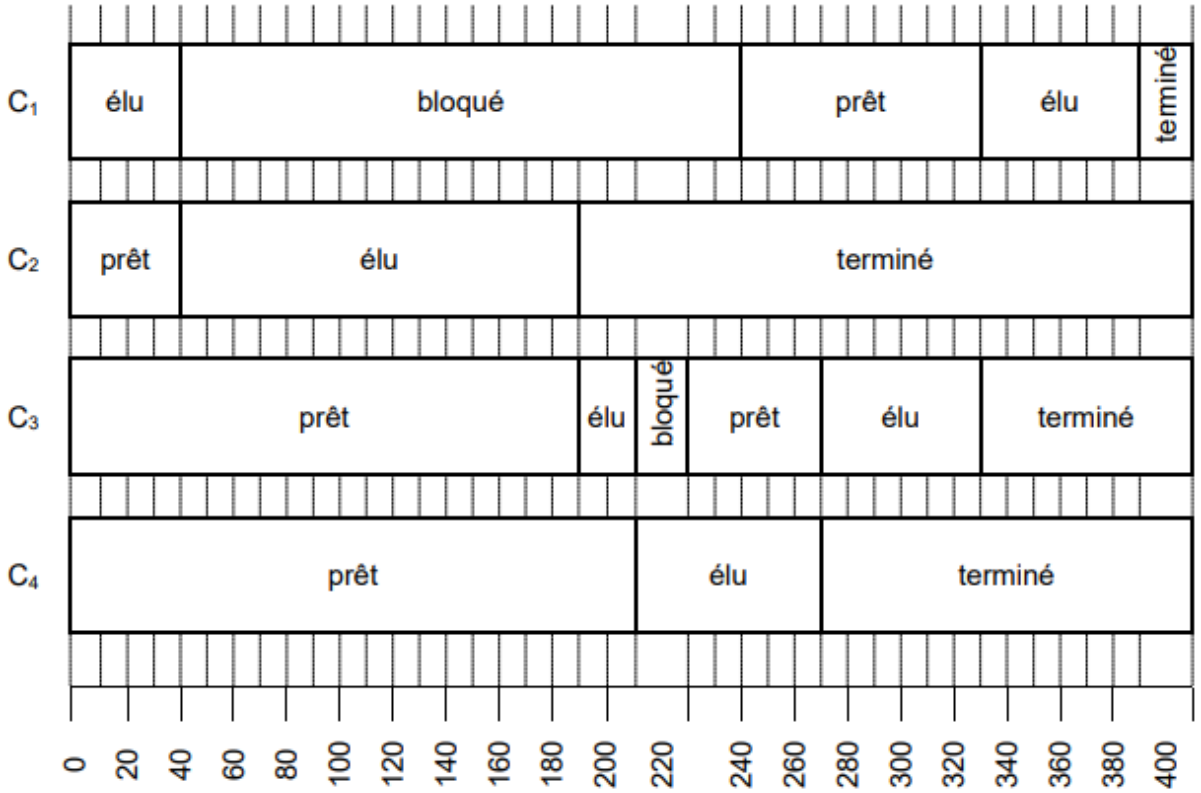
1b



2a

Premier entré, premier sorti

2b



3a

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). P2 attend le fichier_1 avant de pouvoir effectuer les calculs (et donc libérer le fichier_2). Nous avons donc une situation d'interblocage

3b

Il suffit d'inverser les 2 premières actions pour le programme 2 :
Verrouiller fichier_1
Verrouiller fichier_2

Exercice 3

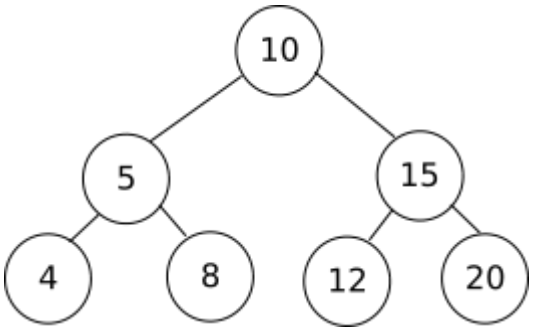
1a

taille de l'arbre = 7

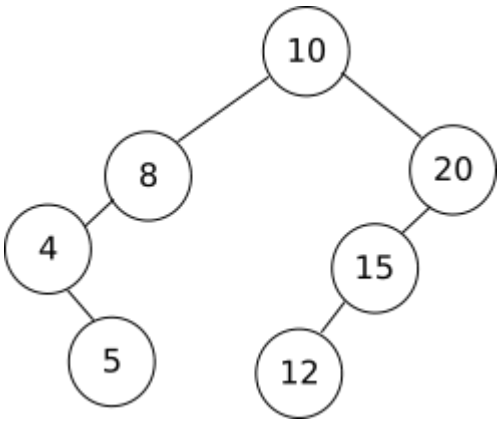
1b

hauteur de l'arbre = 4

2



3



4

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

5

méthode taille de la classe Noeud :

```
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()
```

6a

t_min = 2^{h-1}

6b

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

Exercice 4

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`, le résultat obtenu est `lst[i1] = 3` et `lst[2] = 3,` le code Python proposé ne réalise pas l'échange attendu.

Il faut utiliser une variable temporaire pour que cela fonctionne :

```
temp = lst[i2]
lst[i2] = lst[i1]
lst[i1] = temp
```

2

Les valeurs qui pourront être renvoyées par `randint(0, 10)` sont : 0, 1, 9 et 10

3a

Nous avons un appel récursif avec melange(lst, ind-1). À chaque appel récursif on soustrait 1 au paramètre ind. Au bout d'un certain nombre d'appels récursifs, le paramètre sera égal à 0, les instructions "contenues" dans le "if" (`if ind>0`) ne seront plus exécutées et le programme s'arrêtera.

3b

Pour l'appel initial de la fonction nous avons `ind = n-1`. Pour le premier appel récursif nous avons `ind = n-2`. Pour le dernier appel récursif nous avons `ind = 0`, nous avons donc eu `n-1` appels récursifs.

3c

```
[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
```

3d

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

Exercice 5

1a

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).

1b

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)

2a

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

2b

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

2c

```
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)
```

3a

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

3b

```
def pgsp2(lst):
    somme_max = [lst[0]]
    for i in range (1,len(lst)):
        if somme_max[i-1] <= 0:
            somme_max.append(lst[i])
        else :
            somme_max.append(lst[i]+somme_max[i-1])
    return max(somme_max)
```

3c

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 $O(n^2)$.