Éléments de correction sujet 01 (2022)

Exercice 1

1. La fonction A est une fonction récursive, car elle s'appelle elle-même (`return "a" + A() + "a"`).
Si, par hasard, la fonction choice renvoie systématiquement False, les appels récursifs seraient trop nombreux (dépassement de la taille de la pile), nous aurions alors une erreur d'exécution.

2.

```
def A(n):
    if n <= 0 or choice([True, False]) :
        return "a"
    else:
        return "a" + A(n-1) + "a"
```

Dans le cas où la fonction choice renvoie systématiquement False, nous aurons 50 appels récursifs puisque pour le 51e appel de la fonction A nous aurons n qui sera égal à zéro, le `return "a"` sera donc obligatoirement exécuté (présence du `or` dans le `if`), ce qui entrainera la fin de l'exécution du programme.

3.
- l'appel B(0) renverra systématiquement "bab"
- l'appel B(1) renverra "bab" ou "bbabb"
- l'appel B(2) renverra "bab", "baaab", "bbabb" ou "bbbabbb"

4. a.

```
def regleA(chaine):
    n = len(chaine)
    if n >= 2:
        return chaine[0] == "a" and chaine[n-1] == "a" and
regleA(raccourcir(chaine))
    else:
        return chaine == "a"
```

b.

```
def regleB(chaine):
    n = len(chaine)
    if n >= 2:
        return chaine[0] == "b" and chaine[n-1] == "b" and
(regleA(raccourcir(chaine)) or regleB(raccourcir(chaine)))
    else:
        return False
```

Exercice 2

1.

Numéro du périphérique	Adresse	Opération	Réponse de l'ordonnanceur
0	10	écriture	OK
1	11	lecture	OK
2	10	lecture	ATT
3	10	écriture	ATT
0	12	lecture	OK
1	10	lecture	OK
2	10	lecture	OK
3	10	écriture	ATT

2. Le périphérique 1 ne pourra jamais effectuer sa lecture à l'adresse mémoire 10 puisque juste avant le périphérique 0 aura effectué une écriture à l'adresse mémoire 10.
3.
    a. - tour 1 : 0 peut écrire ; 1 ne peut pas lire
       - tour 2 : 0 ne peut pas écrire ; 1 peut lire
       - tour 3 : 0 peut écrire ; 1 ne peut pas lire
       - tour 4 : 0 peut écrire ; 1 ne peut pas lire
    b. 0 peut écrire trois fois et 1 peut lire une seule fois, seulement un tiers des valeurs écrites par 0 sont lues par 1, soit 33%

4.

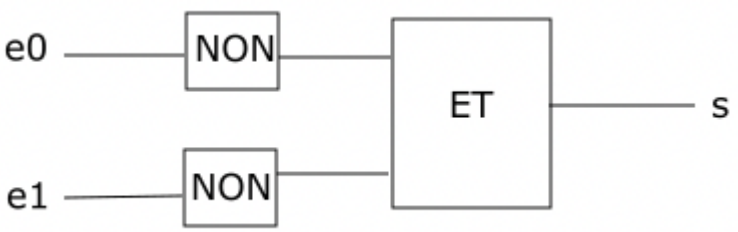
Tour	Numéro du périphérique	Adresse	Opération	Réponse de l'ordonnanceur	ATT_L	ATT_E
1	0	10	écriture	OK	vide	vide
1	1	10	lecture	ATT	(1,10)	vide
1	2	11	écriture	OK	(1,10)	vide
1	3	11	lecture	ATT	(1,10) (3,11)	vide
2	1	10	lecture	OK	(3,11)	vide
2	3	11	lecture	OK	vide	vide
2	0	10	écriture	ATT	vide	(0,10)
2	2	12	écriture	OK	vide	(0,10)
3	0	10	écriture	OK	vide	vide
3	1	10	lecture	ATT	(1,10)	vide
3	2	11	écriture	OK	(1,10)	vide
3	3	12	lecture	OK	(1,10)	vide

5.

a.

b.

c.



Exercice 3

1.
   a.
```sql
SELECT ip, nompage
FROM Visites
```

   b.
```sql
SELECT DISTINCT ip
FROM Visites
```

   c.
```sql
SELECT nompage
FROM Visites
WHERE ip = "192.168.1.91"
```

2.
   a. L'attribut *identifiant* est la clé primaire de la table *Visites*
   b. L'attribut *identifiant* est la clé étrangère de la table *Pings*
   c. Le SGBD va vérifier que l'attribut *identifiant* est unique pour chaque p-upplet de la table Visites. Le SGBD va aussi vérifier que l'attribut *identifiant* de la table Pings correspond bien à un attribut *identifiant* de la table Visites pour chaque p-upplet de la table Pings.

3.
```sql
INSERT INTO Pings
(identifiant, duree)
VALUES
(1534, 105)
```

4.
   a.
```sql
UPDATE Pings
SET duree = 120
WHERE identifiant = 1534
```

   b. Nous ne pouvons pas être certains que les données envoyées par une page web, depuis le navigateur d'un client, via plusieurs requêtes formulées en JavaScript, arrivent au serveur dans l'ordre dans lequel elles ont été émises, car les paquets de données (protocole TCP/IP) n'emprunteront pas forcément le même chemin pour aller du client vers le serveur (ou même certains paquets de données pourront se "perdre" et devront être réémis par le client). Les requêtes HTTP étant constitués de paquets de données, il n'est donc pas possible de garantir l'ordre d'arrivée des requêtes HTTP.
   c. Imaginons que le client envoie 2 requêtes HTTP A et B (le client envoie la requête A puis la requête B). Si le serveur, en recevant la requête B tente de mettre à jour l'entrée correspondant à la requête A alors qu'il n'a pas encore reçu la requête A, cela va entrainer une erreur (tente de mettre à jour une entrée qui n'existe pas).

5. SELECT nompage
   FROM Visites
   JOIN Pings ON Visites.identifiant = Pings.identifiant
   WHERE duree > 60

Exercice 4

1.

```
def est_triee(self):
    if not self.est_vide() :
        e1 = self.depiler()
        while not self.est_vide():
            e2 = self.depiler()
            if e1 > e2 :
                return False
            e1 = e2
    return True
```

2.

a. 3 est plus petit que 4, la valeur renvoyée par l'appel est donc False.
b. état de la pile A : [1, 2]

3.

```
def depileMax(self):
        assert not self.est_vide(), "Pile vide"
        q = Pile()
        maxi = self.depiler()
        while not self.est_vide() :
            elt = self.depiler()
            if maxi < elt :
                q.empiler(maxi)
                maxi = elt
            else :
                q.empiler(elt)
        while not q.est_vide():
            self.empiler(q.depiler())
        return maxi
```

4.

a.

itération 1
- pile B : [9, -7, 8]
- pile q : [4]

itération 2
- pile B : [9, -7]
- pile q : [4, 8]

itération 3
- pile B : [9]
- pile q : [4, 8, -7]

itération 4
- pile B : [ ]
- pile q : [4, 8, -7, 9]

b.
   - pile B : [9, -7, 8, 4]
   - pile q : [ ]

c. Si on utilise une pile C = [19, -7, 8, 12, 4], on obtiendra à la fin la pile suivante : [12, -7, 8, 4]. On peut constater que le 12 ne se situe plus entre le 4 et le 8.

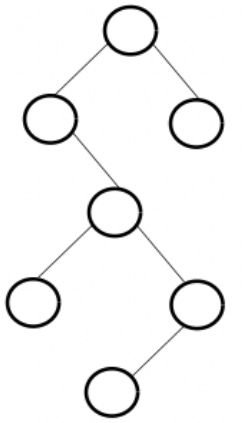
5.

a.
   - avant la ligne 3 : B = [1, 6, 4, 3, 7, 2] et q = [ ]
   - avant la ligne 5 : B = [ ] et q = [7, 6, 4, 3, 2, 1]
   - à la fin : B = [1, 2, 3, 4, 6, 7] et q = [ ]

b. La méthode traiter permet de ranger la pile par ordre croissant (plus grande valeur au sommet de la pile).

Exercice 5

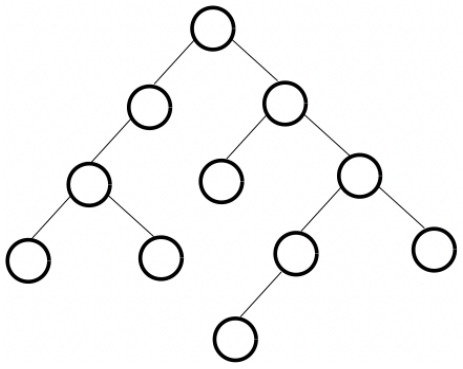
1.
   a. La hauteur de l'arbre est de 2
   b.



2.

```
Algorithme hauteur(A):
  test d'assertion : A est supposé non vide
  si sous_arbre_gauche(A) vide et sous_arbre_droit(A) vide:
    renvoyer 0
  sinon, si sous_arbre_gauche(A) vide:
    renvoyer 1 + hauteur(sous_arbre_droit(A))
  sinon, si sous_arbre_droit(A) vide :
    renvoyer 1 + hauteur(sous_arbre_gauche(A))
  sinon:
    renvoyer 1 + max(hauteur(sous_arbre_gauche(A)),
          hauteur(sous_arbre_droit(A)))
```

3.
   a. Si D est l'arbre vide, la hauteur de R est égale à 1 + hauteur(G) soit 1+2=3, comme la hauteur de R est 4, D ne peut pas être l'arbre vide .
      Comme ni D et ni G sont des arbres vides, hauteur(R) = 1+max(hauteur(G), hauteur(D)), d'où hauteur(R) = 1+max(2,hauteur(D)) avec hauteur(R) = 4, hauteur(D) est obligatoirement égale à 3.
   b.



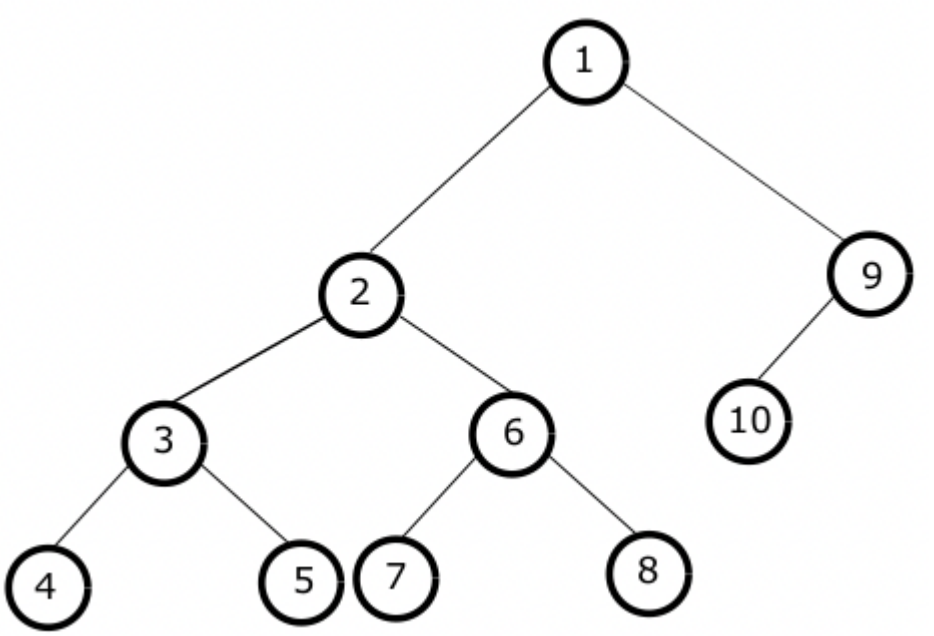
4.

a. L'arbre de la question 1.a. possède 4 nœuds d'où n = 4
Ce même arbre a une hauteur de 2 d'où h = 2
D'où h+1 = 3 et $2^{h+1} - 1 = 2^3 - 1 = 7$
Nous avons bien 4 qui est compris entre 3 et 7, les inégalités sont donc vérifiées.

b. Pour créer un arbre binaire de hauteur h quelconque ayant h+1 nœuds, il suffit de créer un arbre où chaque nœud, a au maximum un enfant.

c. Pour créer un arbre binaire de hauteur h quelconque ayant $2^{h+1} - 1$ nœuds, il suffit de créer un arbre où chaque nœud, qui n'est pas une feuille, a 2 enfants.

5.



6.

```
def fabrique(h, n):
  def annexe(hauteur_max):
    if n == 0 :
      return arbre_vide()
    elif hauteur_max == 0:
      n=n-1
      return arbre(arbre_vide(), arbre_vide())
    else:
      n=n-1
      gauche = annexe(hauteur_max - 1)
      droite = annexe(hauteur_max - 1)
      return arbre(gauche, droite)
  return annexe(h)
```