Exercice 3 (8 points)

Cet exercice porte sur les graphes, les bases de données, les tris, les algorithmes gloutons et la récursivité.

Une association s'occupe d'enfants de 0 à 18 ans. Elle souhaite pouvoir former des groupes d'enfants qui s'entendent durant les activités proposées.

Partie A : base de données

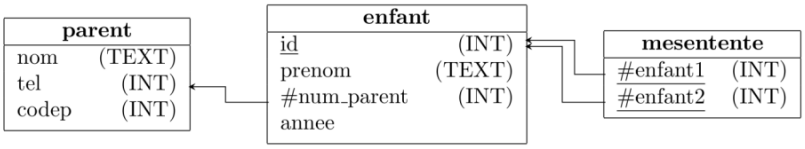


Figure 1. Schéma relationnel de la BDD

On considère la table **enfant** suivante :

enfant			
id	prenom	num_parent	annee
2	'Hawa'	33619911212	2012
3	'Adrien'	33619861232	2013

1. Donner le type pour l'attribut `annee` de la table **enfant**.

Le type pour l'attribut **annee** de la table **enfant** est INT (**INTEGER)**.

2. Expliquer quelle contrainte de domaine supplémentaire serait pertinente pour cet attribut `annee`.

Une contrainte de domaine supplémentaire pertinente pour l'attribut **annee** serait de s'assurer que l'année est comprise entre une année minimale et l'année actuelle.
Que la date contienne 4 chiffres…

3. Donner un exemple d'attribut de la table **enfant** qui suit une contrainte de référence.

Un exemple d'attribut de la table **enfant** qui suit une contrainte de référence est **num_parent**, qui fait référence à l'attribut **tel** de la table **parent**.

4. En expliquant ce choix, proposer une clef primaire pour la table **parent**.

Une clef primaire pertinente pour la table **parent** serait l'attribut **tel**, car il est unique pour chaque parent (Numéro de portable). Par contre on ne gère pas les deux parents ni les parents séparés.

Suite à une mauvaise saisie, le véritable téléphone d'un parent (33619782812) a été transformé en 33600782812. On souhaite corriger cette anomalie avec la requête suivante, mais elle lève une erreur.

```
UPDATE parent SET tel = 33619782812 WHERE tel = 33600782812;
```

5. Expliquer pourquoi la requête proposée lève une erreur.

La requête proposée lève une erreur car elle tente de mettre à jour un numéro de téléphone qui est utilisé comme clé étrangère dans la table **enfant**. Pour éviter cette erreur, il faut d'abord mettre à jour les références dans la table **enfant**.

6. Recopier et compléter alors cette suite de commandes qui permet de changer le numéro de téléphone d'un parent du parent de nom 'Bauges' habitant au code postal 73340 et ayant pour téléphone erroné 33600782812 au lieu de son véritable téléphone 33619782812 :

```
INSERT INTO parent VALUES ('Bauges', 33619782812, 73340);
UPDATE enfant SET num_parent = 33619782812 WHERE num_parent = 33600782812;
DELETE FROM parent WHERE tel = 33600782812;
```

7. En considérant la table `enfant` fournie, donner le résultat de cette requête `SQL`.

```
SELECT prenom
FROM enfant
WHERE annee < 2014
ORDER BY annee;
```

Nakamura
Hawa
Kian
Adrien

enfant

id	prenom	num_parent	annee
2	'Hawa'	33619911212	2012
3	'Adrien'	33619861232	2013
6	'Kian'	33619834521	2012
8	'Gabin'	33619847852	2014
12	'Nakamura'	33619732453	2009

8. Proposer une requête qui renvoie les prénoms, par ordre alphabétique, des enfants inscrits pour le parent dont le numéro de téléphone est `3619861122`.

```
SELECT prenom
FROM enfant
WHERE num_parent = 3619861122
ORDER BY prenom;
```

9. Proposer une requête qui liste les identifiants et prénoms des enfants dont le parent habite dans la ville de code postal `38520`.

```
SELECT e.id, e.prenom
FROM enfant e
JOIN parent p ON e.num_parent = p.tel
WHERE p.codep = 38520;
```

Partie B : graphes et algorithmique

```
1 g1 = {    'Elise': ['Octavie', 'Virgile'],
2           'Octavie': ['Elise', 'Pierre', 'Virgile'],
3           'Pierre': ['Octavie', 'Raphael'],
4           'Raphael': ['Pierre', 'Virgile'],
5           'Sixtine': [],
6           'Virgile': ['Elise', 'Octavie', 'Raphael']
7           }
```

10. Expliquer pourquoi la situation décrite ne nécessite qu'un graphe non orienté.

La situation décrite ne nécessite qu'un graphe non orienté car la mésentente entre deux enfants est soit une relation symétrique les deux enfants ne s'entendent pas ou si le problème vient d'un seul enfant on ne prendra pas le risque le mettre l'autre avec lui.

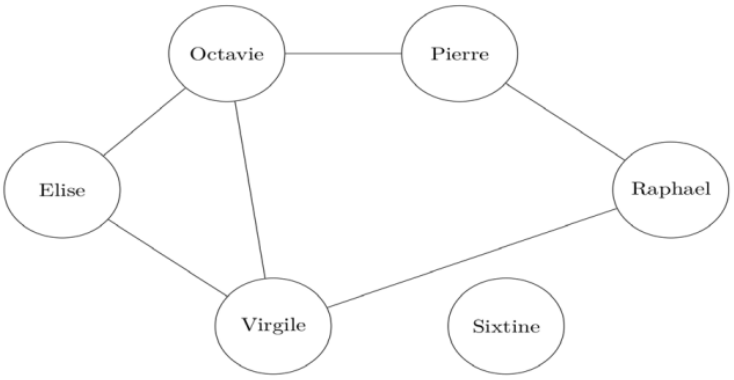
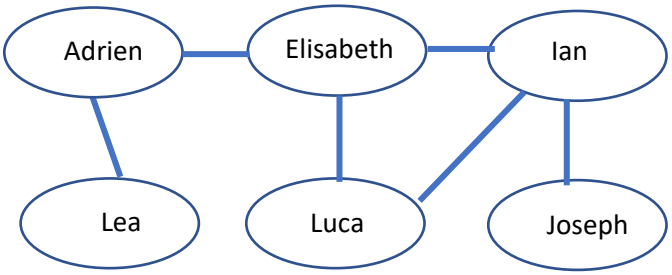


Figure 2. Graphe `g1`

11. Dessiner le graphe `g2` défini ci-dessous.

```
1 g2 = {    'Adrien': ['Elisabeth', 'Lea'],
2           'Elisabeth': ['Adrien', 'Ian', 'Luca'],
3           'Ian': ['Elisabeth', 'Joseph', 'Luca'],
4           'Joseph': ['Ian'],
5           'Lea': ['Adrien'],
6           'Luca': ['Elisabeth', 'Ian']
7     }
```



12. Écrire une fonction `degre`, qui prend en arguments un dictionnaire `g` représentant un graphe et une chaîne de caractères `s` représentant un sommet du graphe, et qui renvoie le degré du sommet `s` dans `g`. On rappelle que le degré d'un sommet est le nombre d'arêtes issues de ce sommet.

```
def degre(g, s):
    return len(g[s])
```

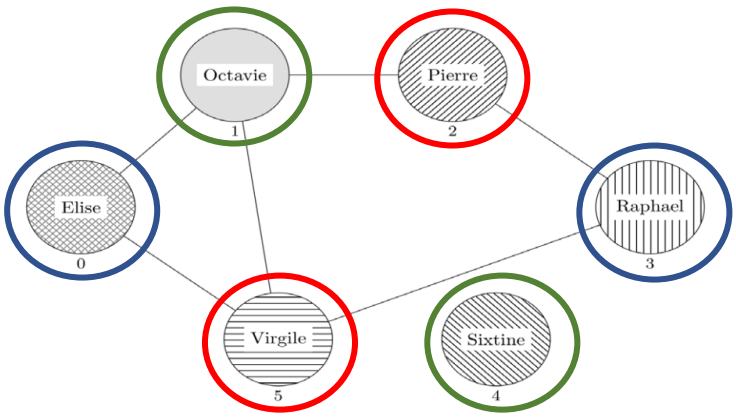
13. Recopier et compléter les lignes 7 à 10 de la fonction `sommets_tries`, qui prend en paramètre un dictionnaire `g` représentant un graphe, et qui renvoie la liste des sommets du graphe triés dans l'ordre décroissant de leur degré.

```
1 def sommets_tries(g):
2     sommets = [sommet for sommet in g]
3     n = len(sommets)
4     for i in range(1, n):
5           sommet_courant = sommets[i]
6           j = i-1
7           while j >= 0 and degre(g, sommets[j]) < degre(g,sommet_courant):
8                 sommets[[j + 1] = sommets[j]
9                 j = j - 1
10          sommets[j + 1] = sommet_courant
11    return sommets
```

14. Préciser le tri utilisé dans la question précédente, ainsi que son coût d'exécution en temps dans le pire des cas selon le nombre nn de sommets (constant, logarithmique soit en $\log_2(nn)$, linéaire soit en nn, quasi-linéaire soit en $nn\log_2(nn)$, quadratique soit en nn^2, cubique soit en nn^3, exponentiel soit en 2^{nn}, …). On fait l'hypothèse pour cette question que la fonction `degre` est de coût constant.

Le tri utilisé dans la question précédente est le tri par insertion. Son coût d'exécution en temps dans le pire des cas est quadratique, soit en $O(n^2)$.

15. Recopier et colorer le graphe g1 en n'utilisant que trois couleurs (0, 1 et 2).



On dispose de la fonction `plus_petite_couleur_hors_voisins`, qui prend en paramètres

- un dictionnaire `g` représentant un graphe,
- un dictionnaire de couleurs `dc` dont les clés sont des sommets de `g`,
- une chaîne de caractères `s` correspondant à un sommet de `g`, et qui renvoie le plus petit numéro non utilisé dans `dc` par les voisins du sommet `s`.

```
1 def couleurs_voisins(g, dc, s):
2     return [dc[v] for v in g[s]]
3
4 def plus_petite_couleur_hors_voisins(g, dc, s):
5     couleur = 0
6     n = len(g)
7     cvoisins = couleurs_voisins(g, dc, s)
8     while couleur < n:
9           if couleur not in cvoisins:
10                return couleur
11          couleur = couleur + 1
12    return couleur # au cas où len(dc) = 0
```

16. Recopier et compléter la procédure qui permet de colorer le graphe en modifiant le dictionnaire `dc` pour qu'il associe finalement à chaque sommet de `g` sa couleur.

```
1 def colorer_graphe(g, dc):
2     # Pré-condition : les clés de dc sont les sommets
3     # de g, et les valeurs de dc sont toutes à -1
4     for s in dc:
5           couleur = plus_petite_couleur_hors_voisins(g, dc, s)
6           dc[s] = couleur
```

17. Recopier et compléter le code de la fonction `welsh_powell` donné ci-après. Cette fonction prend en paramètre un dictionnaire `g` correspondant à un graphe, et le colore selon l'algorithme de Welsh-Powell (c'est-à-dire qu'elle renvoie le dictionnaire des couleurs associé).

18.

On pourra s'inspirer de la fonction `colorer_graphe` donnée ci-dessus et utiliser la fonction `sommets_tries`.

```
1 def welsh_powell(g):
2     # initialisation à -1 pour tous les sommets dans le dictionnaire dc
3     dc = {sommet: -1 for sommet in g} # possiblement plusieurs lignes
      #dc = {}
      #for sommet in g1:
            #dc[sommet] = -1
4     # coloration en suivant l'approche de Welsh-Powell
5     for s in sommets_tries(g):
6           couleur = plus_petite_couleur_hors_voisins(g, dc, s)
7           dc[s] = couleur
8     return dc
```