EXERCICE 2 (6 points)

Cet exercice porte sur la programmation objet, la récursivité, les arbres binaires et les systèmes d'exploitation.

Dans cet exercice, on travaille dans un environnement Linux. On considère l'arborescence de fichiers de la figure 1.

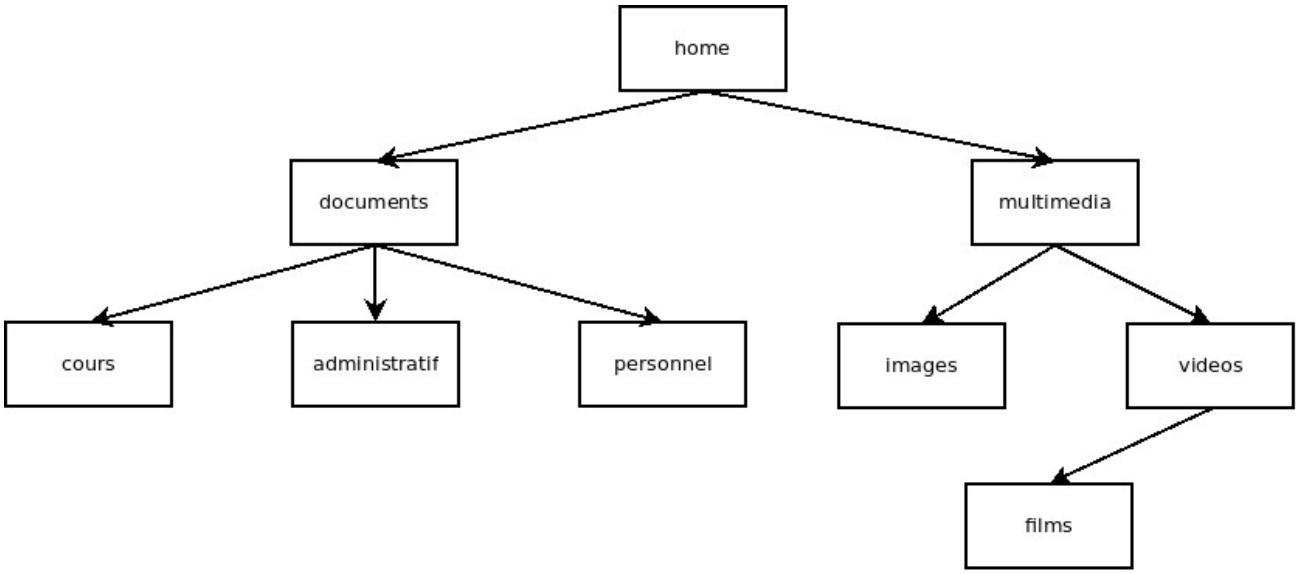

Figure 1. Arborescence de fichiers

Partie A

1. Le répertoire courant est `home`. Donner une commande permettant de connaître le contenu du dossier `documents`.

On suppose que l'on se trouve dans le dossier `cours` et que l'on exécute la commande `mv ../../multimedia /home/documents`

2. Indiquer la modification que cela apporte dans l'arborescence de la figure 1.

On considère le code suivant :

```
1 class Arbre:
2     def __init__(self, nom, g, d):
3         self.nom = nom
4         self.gauche = g
5         self.droit = d
6
7     def est_vide(self):
8         return self.gauche is None and self.droite is None
9
```

```
10      def parcours(self):
11          print(self.nom)
12          if self.gauche != None:
13              self.gauche.parcours()
14          if self.droit != None:
15              self.droit.parcours()
```

3. Donner une raison qui justifie que le code précédent ne permet pas de modéliser l'arborescence de fichiers de la figure 1.

4. Donner le nom du parcours réalisé par le code précédent.

5. Donner la liste des dossiers dans l'ordre d'un parcours en largeur de l'arborescence. On ne demande pas d'écrire ce parcours en Python.

Partie B

Pour pouvoir modéliser l'arborescence de fichiers de la figure 1, on propose l'implémentation suivante. L'attribut `fils` est une variable de type `list` contenant tous les dossiers fils. Cette liste est vide dans le cas où le dossier est vide.

```
1 class Dossier:
2     def __init__(self, nom, liste):
3         self.nom = nom
4         self.fils = liste # liste d'objets de la classe Dossier
```

6. Écrire le code Python d'une méthode `est_vide` qui renvoie `True` lorsque le dossier est vide et `False` sinon.

7. Écrire le code Python permettant d'instancier une variable `var_multimedia` de la classe `Dossier` représentant le dossier `multimedia` de la figure 1. Attention : cela nécessite d'instancier tous les nœuds du sous-arbre de racine `multimedia`.

8. Recopier et compléter sur votre copie le code Python de la méthode `parcours` suivante qui affiche les noms de tous les descendants d'un dossier en utilisant l'ordre préfixe.

```
1 def parcours(self):
2     print(...)
3     for f in ...:
4         ...
```

9. Justifier que cette méthode `parcours` termine toujours sur une arborescence de fichiers.

10. Proposer une modification de la méthode `parcours` pour que celle-ci effectue plutôt un parcours suffixe (ou postfixe).

11. Expliquer la différence de comportement entre un appel à la méthode `parcours` de la classe `Dossier` et une exécution de la commande UNIX `ls`.

On considère la variable `var_videos` de type `Dossier` représentant le dossier `videos` de la figure 1. On souhaite que le code Python `var_videos.mkdir("documentaires")` crée un dossier `documentaires` vide dans le dossier `var_videos`.

12. Écrire le code Python de la méthode `mkdir`.

13. Écrire en Python une méthode `contient(self, nom_dossier)` qui renvoie `True` si l'arborescence de racine `self` contient au moins un dossier de nom `nom_dossier` et `False` sinon.

14. Avec l'implémentation de la classe `Dossier` de cette partie, expliquer comment il serait possible de déterminer le dossier parent d'un dossier donné dans une arborescence donnée. On attend ici l'idée principale de l'algorithme décrite en français. On ne demande pas d'implémenter cet algorithme en Python.

15. Proposer une modification dans la méthode `__init__` de la classe `Dossier` qui permettrait de répondre à la question précédente beaucoup plus efficacement et expliquer votre choix.

EXERCICE 3 (8 points)

Cet exercice porte sur le codage binaire, les bases de données relationnelles et les requêtes SQL.

Cet exercice est composé de deux parties peu dépendantes entre elles.

Lorsqu'il y a un accident, les pompiers essaient d'intervenir sur les lieux le plus rapidement possible avec les équipes et le matériel adéquats. On s'intéresse à l'étude d'un système informatique simplifié permettant de répondre à certaines de leurs problématiques.

Chaque pompier possède des aptitudes opérationnelles qui lui permettent de tenir un rôle. Lors du départ d'un véhicule (on parle d'agrès), il faut *a minima* un conducteur et un chef d'agrès.

Pour simplifier, on considère que

- un Véhicule Tout Usage (VTU) ne requiert que le duo conducteur et chef d'agrès, donc deux pompiers ;
- un Véhicule de Secours et d'Assistance aux Victimes (VSAV) requiert, en plus du duo conducteur et chef d'agrès, un équipier, donc trois pompiers ;
- un Fourgon Pompe Tonne (FPT) requiert, en plus du duo conducteur et chef d'agrès, un chef d'équipe et un équipier, donc quatre pompiers.

On souhaite entrer dans une base de données l'ensemble de ces informations afin de pouvoir conserver un historique des interventions.

Partie A – Encodage binaire

Afin de gagner de la place mémoire, on décide de coder l'ensemble des aptitudes sur un entier de 8 bits plutôt que d'écrire en toutes lettres "équipier", "chef d'équipe", etc. Cet ensemble d'aptitudes formera la qualification du pompier considéré.

- Un personnel non formé est codé par 0 ;
- l'aptitude "équipier" est codée par le bit de rang 0 (celui de poids le plus faible), soit 2^0 ;
- l'aptitude "chef d'équipe" est codée par le bit de rang 1, soit 2^1 ;
- l'aptitude "chef d'agrès" est codée par le bit de rang 2, soit 2^2 ;
- enfin, l'aptitude "conducteur" est codée par le bit de rang 3, soit 2^3.

Remarque : un chef d'équipe étant nécessairement équipier, on lui ajoute cette aptitude dans son codage. De même, un chef d'agrès est nécessairement un chef d'équipe et un équipier : on lui ajoute ces aptitudes dans son codage.