EXERCICE 2 (6 points)

Cet exercice porte sur les thèmes suivants : file, pile et arbre.

Chaque réclamation d'un utilisateur concernant un incident sur l'informatique du Lycée provoque la création d'un ticket qui contient les informations suivantes :

- Le numéro de ticket : un identifiant unique croissant ordonné par date de création du ticket ;
- Le type de matériel concerné : soit **PC** soit **Projecteur ;**
- La priorité de la réclamation : soit **Moyenne** soit **Elevée**.

Partie A

Lorsqu'un ticket est créé, il est :

- D'abord enfilé dans la file **tickets_avant_affectation**;
- Puis, après assignation à une équipe, il est défilé de la file **tickets_avant_affectation** et est enfilé soit dans la file **tickets_PC** soit dans la file **tickets_projecteur** selon le type de matériel concerné par la réclamation.

On dispose pour cela d'une structure de données abstraite File avec les fonctions suivantes :

- **`creerFile()`** qui renvoie une file vide ;
- **`estFileVide(file)`** qui renvoie **True** si **file** est vide et **False** sinon ;
- **`enfiler(file, element)`** qui ajoute **element** en queue de **file** ;
- **`defiler(file)`** qui retire l'élément en tête de **file** et le renvoie.

On considère le programme ci-après :

```
    def assignerTickets(file):
1       file_PC = creerFile()
2       file_projecteur = creerFile()
3       ... (plusieurs lignes)
4       return file_PC, file_projecteur
5
6
7   tickets_avant_affectation = creerFile()
8
9   ticket_1 = {'id': 1, 'Type': 'PC', 'Prio': 'Moyenne'}
10  enfiler(tickets_avant_affectation, ticket_1)
11  enfiler(tickets_avant_affectation, {'id': 2, 'Type': 'PC', 'Prio': 'Elevée'})
12  enfiler(tickets_avant_affectation, {'id': 3, 'Type': 'Projecteur', 'Prio': 'Moyenne'})
13
14  print('File des tickets avant affectation : ', tickets_avant_affectation)
15
16  file_PC, file_projecteur = assignerTickets(tickets_avant_affectation)
17
18  print('File des tickets affectés à l\'équipe PC : ', file_PC)
19  print('File des tickets affectés à l\'équipe projecteur : ', file_projecteur)
20
```

1. Donner le type de la variable **`ticket_1`** représentant le premier ticket créé et décrire les informations associées en utilisant le vocabulaire associé à son type. 0,25 pt

Dictionnaire avec trois éléments (entrées) :
1er élément : la clé est 'id' et sa valeur 1 ;
2ème élément : la clé est 'Type' et sa valeur 'PC' ;
3ème élément : la clé est 'Prio' et sa valeur 'Moyenne'.

2. Donner l'affichage en console après exécution des **`print`** en lignes 15, 19 et 20. 0,5 pt

File des tickets avant affectation : [{'id': 1, 'Type': 'PC', 'Prio': 'Moyenne'}, {'id': 2, 'Type': 'PC', 'Prio': 'Elevée'}, {'id': 3, 'Type': 'Projecteur', 'Prio': 'Moyenne'}]
File des tickets affectés à l'équipe PC : [{'id': 1, 'Type': 'PC', 'Prio': 'Moyenne'}, {'id': 2, 'Type': 'PC', 'Prio': 'Elevée'}]

File des tickets affectés à l'équipe projecteur : [{'id': 3, 'Type': 'Projecteur', 'Prio': 'Moyenne'}]

3. Recopier et compléter la fonction **`assignerTickets`** qui construit et retourne un tuple avec une file pour l'équipe PC et une file pour l'équipe projecteur selon le type de la réclamation objet du ticket. 0,75 pt

```
def assignerTickets(file):
    file_PC = creerFile()
    file_projecteur = creerFile()
    while not estFileVide(file):
        ticket = defiler(file)
        if ticket['Type'] == 'PC':
            enfiler(file_PC, ticket)
        elif ticket['Type'] == 'Projecteur':
            enfiler(file_projecteur, ticket)
    return file_PC, file_projecteur
```

On veut réorganiser l'ordre des tickets dans une file d'équipe pour que les tickets de priorité élevée soient placés avant les tickets de priorité moyenne.

On procède ainsi :

- 1ère étape : on défile entièrement la file d'équipe et on empile les tickets défilés, selon leur priorité, soit dans une pile priorité élevée, soit dans une pile priorité moyenne ;
- 2ème étape : on renverse chacune des deux piles ;
- 3ème étape : on dépile d'abord la pile priorité élevée et on enfile les tickets dépilés dans la file équipe, puis on dépile la pile priorité moyenne et on enfile les tickets dépilés dans la file équipe.

Par exemple, si **`file_PC`** contient initialement : **1M 2E 4E 5M 6M 7E**, avec **1M** représentant un ticket d'identifiant 1 et de priorité moyenne, et **2E** représentant un ticket d'identifiant 2 et de priorité élevée, on aura à la fin de chaque étape :

	File_PC	Pile priorité élevée	Pile priorité moyenne
1ère étape	**Vide**	**7E 4E 2E**	**6M 5M 1M**
2ème étape	**Vide**	**2E 4E 7E**	**1M 5M 6M**
3ème étape	**2E 4E 7E 1M 5M 6M**	**Vide**	**Vide**

On dispose pour cela, en plus de la structure de file précédente, d'une structure de données abstraite Pile avec les fonctions suivantes :

- **`creerPile()`** qui renvoie une pile vide ;
- **`estPileVide(pile)`** qui renvoie **`True`** si **`pile`** est vide et **`False`** sinon ;
- **`empiler(pile, element)`** qui ajoute **`element`** au sommet de **`pile`** ;
- **`depiler(pile)`** qui retire le sommet de **`pile`** et le renvoie.

On complète le programme précédent par le code ci-après :

```
22
23  def renverserPile(pile):
24      pile_r = creerPile()
25      ... (plusieurs lignes)
26      return pile_r
27
28
29  def reorganiserFilePriorites(file):
30
31      # étape 1
32      pile_moyenne = creerPile()
33      pile_elevee = creerPile()
34      ... (plusieurs lignes)
35
36      # étape 2
37      ... (plusieurs lignes)
38
39      # étape 3
40      ... (plusieurs lignes)
41
42
43  enfiler(file_PC, {'id': 4, 'Type': 'PC', 'Prio': 'Elevée'})
44  enfiler(file_PC, {'id': 5, 'Type': 'PC', 'Prio': 'Moyenne'})
45  enfiler(file_PC, {'id': 6, 'Type': 'PC', 'Prio': 'Moyenne'})
46  enfiler(file_PC, {'id': 7, 'Type': 'PC', 'Prio': 'Elevée'})
47
48  print(file_PC)
49  reorganiserFilePriorites(file_PC)
50  print(file_PC)
```

4. Donner l'affichage en console après exécution des **`print`** en lignes 48 et 50. 0,25 pt

[{'id': 1, 'Type': 'PC', 'Prio': 'Moyenne'}, {'id': 2, 'Type': 'PC', 'Prio': 'Elevée'}, {'id': 4, 'Type': 'PC', 'Prio': 'Elevée'}, {'id': 5, 'Type': 'PC', 'Prio': 'Moyenne'}, {'id': 6, 'Type': 'PC', 'Prio': 'Moyenne'}, {'id': 7, 'Type': 'PC', 'Prio': 'Elevée'}]

[{'id': 2, 'Type': 'PC', 'Prio': 'Elevée'}, {'id': 4, 'Type': 'PC', 'Prio': 'Elevée'}, {'id': 7, 'Type': 'PC', 'Prio': 'Elevée'}, {'id': 1, 'Type': 'PC', 'Prio': 'Moyenne'}, {'id': 5, 'Type': 'PC', 'Prio': 'Moyenne'}, {'id': 6, 'Type': 'PC', 'Prio': 'Moyenne'}]

5. Recopier et compléter la fonction **`renverserPile`** qui renvoie une pile renversée par rapport à celle passée en argument. 0,75 pt

```
def renverserPile(pile):
    pile_r = creerPile()
    while not estPileVide(pile):
        empiler(pile_r,depiler(pile))
    return pile_r
```

6. Recopier et compléter la fonction **`reorganiserFilePriorites`** qui modifie la file passée en argument pour prioriser les tickets à priorité élevée à l'aide des 3 étapes décrites plus haut. 0,75 pt

```
def reorganiserFilePriorites(file):

    # étape 1
    pile_moyenne = creerPile()
    pile_elevee = creerPile()
    while not estFileVide(file):
        ticket = defiler(file)
        if ticket['Prio'] == 'Moyenne':
            empiler(pile_moyenne, ticket)
        elif ticket['Prio'] == 'Elevée':
            empiler(pile_elevee, ticket)


    # étape 2
    pile_moyenne = renverserPile(pile_moyenne)
    pile_elevee = renverserPile(pile_elevee)


    # étape 3
    while not estPileVide(pile_elevee):
            enfiler(file, depiler(pile_elevee))
    while not estPileVide(pile_moyenne):
            enfiler(file, depiler(pile_moyenne))
```

Partie B

Une fois les tickets traités et clôturés, ils sont insérés dans un Arbre Binaire de Recherche (ABR).
On dispose des classes **`Ticket`**, **`ABR`** et **`Noeud`** représentant respectivement un ticket, un ABR et un nœud de l'ABR :

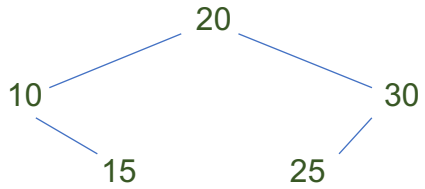
```
class Ticket:
    def __init__(self, id, type, prio, duree):
        self.id = id
        self.type = type
        self.prio = prio
        self.duree = duree


class ABR:
    def __init__(self):
        self.racine = None

    def ajouteTicket(self, ticket):
        if self.racine == None:
            self.racine = Noeud(ticket)
        else :
            self.racine.ajouteTicket(ticket)

    def dureeTotale(self, type, prio):
        if self.racine == None:
            return 0
        else:
            return self.racine.dureeTotale(type, prio)


class Noeud:
    def __init__(self, ticket, gauche = None, droite = None):
        self.ticket = ticket
        self.gauche = gauche
        self.droite = droite

    def ajouteTicket(self, ticket):
        if ticket.id ... self.ticket.id:
            if self.gauche ... :
                self.gauche = Noeud(ticket)
            else:
                ...
        else:
            if self.droite ... :
                self.droite = Noeud(ticket)
            else:
                ...

    def dureeTotale(self, type, prio):
        ... (plusieurs lignes)

```

Chaque nœud de l'ABR contient un ticket qui est caractérisé par un identifiant, tel que :

- Chaque nœud du sous-arbre gauche a un ticket d'identifiant inférieur ou égale à celui du nœud considéré
- Chaque nœud du sous-arbre droit a un ticket d'identifiant strictement supérieur à celui du nœud considéré.

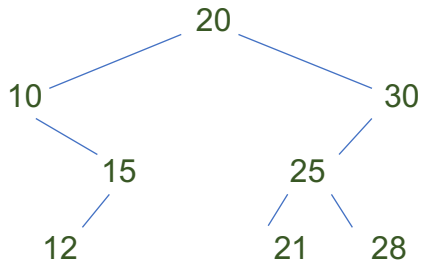
7. Représenter l'ABR, uniquement avec les identifiants des tickets, après exécution du code ci-après : 0,5 pt

```
48   # création de l'arbre
49  arbre = ABR()
50
51  # création du ticket_20, instance de la classe Ticket
52  ticket_20 = Ticket(20, 'PC', 'Moyenne', 10)
53  # et ajout du ticket_20 à l'arbre
54  arbre.ajouteTicket(ticket_20)
55
56  # ajouts de tickets à l'arbre
57  arbre.ajouteTicket(Ticket(10, 'PC', 'Elevée', 30))
58  arbre.ajouteTicket(Ticket(30, 'Projecteur', 'Elevée', 5))
59  arbre.ajouteTicket(Ticket(15, 'PC', 'Moyenne', 20))
60  arbre.ajouteTicket(Ticket(25, 'PC', 'Elevée', 10))
```



8. Recopier votre ABR de la question précédente, sans le modifier, et le compléter pour construire l'ABR résultant de l'exécution du code supplémentaire ci-après : 0,25 pt

```
61
62  # ajouts complémentaires de tickets à l'arbre
63  arbre.ajouteTicket(Ticket(28, 'Projecteur', 'Elevée', 10))
64  arbre.ajouteTicket(Ticket(21, 'PC', 'Moyenne', 40))
65  arbre.ajouteTicket(Ticket(12, 'PC', 'Elevée', 30))
66
```



9. Recopier et compléter la méthode **`ajouteTicket`** de la classe **`Noeud`** qui ajoute **`ticket`** à l'ABR. 1 pt

```
def ajouteTicket(self, ticket):
        if ticket.id <= self.ticket.id:
            if self.gauche is None :
                self.gauche = Noeud(ticket)
            else:
                self.gauche.ajouteTicket(ticket)
        else:
            if self.droite is None :
                self.droite = Noeud(ticket)
            else:
                self.droite.ajouteTicket(ticket)
```

10. Recopier et compléter la méthode **`dureeTotale`** de la classe **`Noeud`** qui retourne la somme des durées de traitement de tous les tickets de l'arbre qui ont le type et la priorité indiqués en paramètres. 1 pt

Exemples :

```
# résultats d'appels à la méthode dureeTotale
>>> print(arbre.dureeTotale('PC', 'Elevée'))
    70
>>> print(arbre.dureeTotale('Projecteur', 'Moyenne'))
    0
>>> print(arbre.dureeTotale('PC', 'Moyenne'))
    70
>>> print(arbre.dureeTotale('Projecteur', 'Elevée'))
    15
```

```
def dureeTotale(self, type, prio):
  if self.gauche is None and self.droite is None:
      if self.ticket.type == type and self.ticket.prio == prio:
          return self.ticket.duree
      else:
          return 0
  elif self.gauche is None:
      if self.ticket.type == type and self.ticket.prio == prio:
          return self.ticket.duree + self.droite.dureeTotale(type, prio)
      else:
          return self.droite.dureeTotale(type, prio)
  elif self.droite is None:
      if self.ticket.type == type and self.ticket.prio == prio:
          return self.ticket.duree + self.gauche.dureeTotale(type, prio)
      else:
          return self.gauche.dureeTotale(type, prio)
  else:
      if self.ticket.type == type and self.ticket.prio == prio:
          return self.ticket.duree + self.gauche.dureeTotale(type, prio)+ \
                                                  self.droite.dureeTotale(type, prio)
      else:
          return self.gauche.dureeTotale(type, prio)+ self.droite.dureeTotale(type, prio)
```

EXERCICE 3 (6 points)

Principaux thèmes abordés : réseaux de télécommunications et graphe.

Le réseau local A est relié au réseau local B par l'intermédiaire d'une série de routeurs $(R1, R2, R3, R4, R5, R6)$ avec l'architecture représentée ci-après :

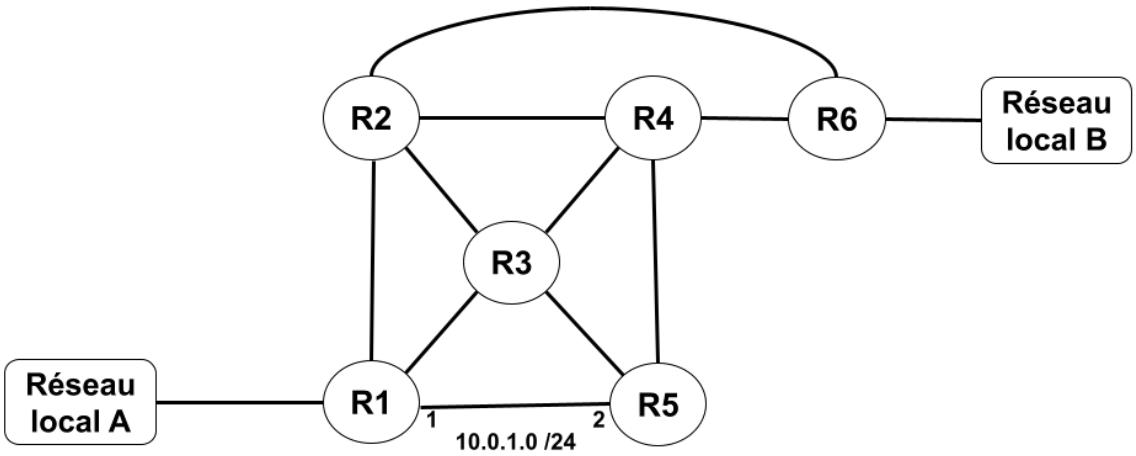


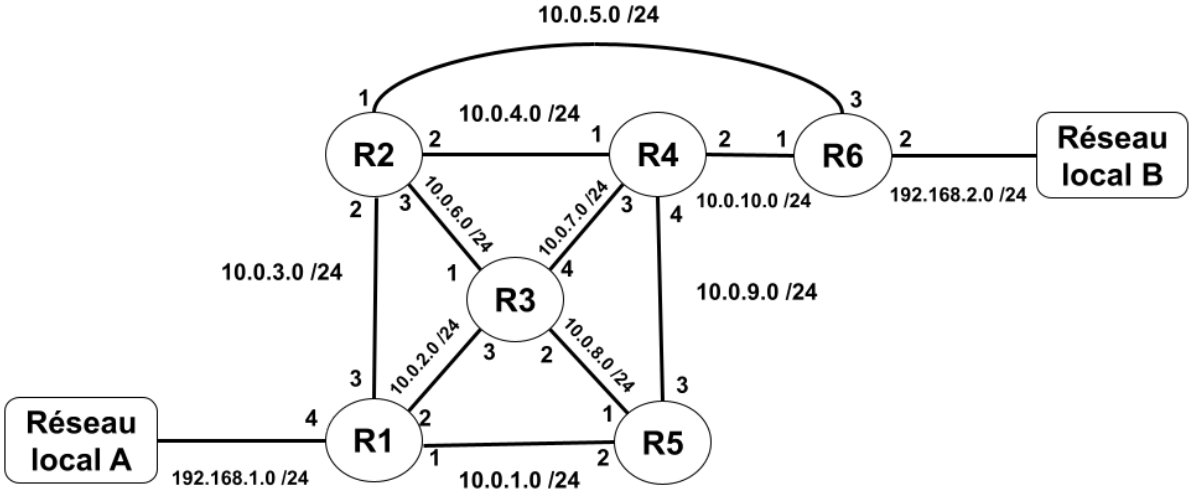
schéma 1. Architecture réseau

Partie A

1. Recopier le réseau et compléter chaque liaison par une adresse réseau et chaque interface de routeur par la partie machine de son adresse, comme pour la liaison $R1 - R5$ (cf : schéma 1. ci-dessus) où : 0,75 pt

- **L'adresse réseau** est 10.0.1.0 ;
- **Le masque réseau** est à 1 sur les 24 premiers bits et à 0 les 8 derniers bits, donc les trois premiers octets représentent la partie réseau (10.0.1) et le dernier octet représente la partie machine (0) ; cela signifie donc que le masque vaut 255.255.255.0.
- **La partie machine** de l'adresse de l'interface de $\boldsymbol{R1}$ est 1 ;
- **La partie machine** de l'adresse de l'interface de $\boldsymbol{R5}$ est 2.

Tout adressage cohérent par exemple :



2. En utilisant l'adressage réseau choisi, donner un affichage possible d'une commande **`traceroute`** lancée depuis le terminal d'une machine du réseau local *A* vers l'adresse IP d'une machine du réseau local *B*. 0,5 pt

Adresses ip traversés cohérentes avec l'adressage choisie, avec notre exemple, on peut avoir :

192.168.1.1
192.168.1.4
10.0.3.2
10.0.5.3
192.168.2.1

Dans un premier temps, on utilise le protocole de routage **RIP** (Routing Information Protocol).
On rappelle que dans ce protocole, la métrique de la table de routage correspond au nombre de routeurs à traverser pour atteindre la destination.

3. Donner la table de routage de $R1$, après la convergence, en indiquant pour chaque route : 0,5 pt

- Le nom du routeur destination ;
- Le nom du routeur passerelle éventuel ;
- Le nombre de sauts pour atteindre le routeur destination.

Destination	Passerelle	Nombre sauts
R1		0
R2	R2	1
R3	R3	1
R4	R2 ou R3 ou R5	2
R5	R5	1
R6	R2	2

4. Le routeur $R2$ tombe en panne. Donner la table de routage de $R1$, après la convergence, sous le même format qu'à la question précédente. 0,5 pt

Destination	Passerelle	Nombre sauts
R1		0
R2		16 ou infini
R3	R3	1
R4	R3 ou R5	2
R5	R5	1
R6	R3 ou R5	3

Le routeur $R2$ est de nouveau fonctionnel. Dans la suite de cet exercice, on utilise le protocole de routage **OSPF** (Open Shortest Path First). On rappelle que dans ce protocole, la métrique de la table de routage correspond à la somme des coûts des liaisons à traverser pour atteindre la destination avec, pour chaque liaison :

$$coût = \frac{10^8}{d}$$ où d est la bande passante de la liaison en bits/s

Le réseau est constitué de 3 types de liaisons de communication :

- Fibre optique avec un débit de 1 Gbits/s ;
- Fast-ethernet avec un débit de 100 Mbits/s ;
- Ethernet avec un débit de 10 Mbits/s.

Le type des différentes liaisons inter-routeurs de notre réseau sont les suivantes :

Fibre optique	Fast-ethernet	Ethernet
$R1-R3$	$R1-R2$	$R2-R6$
$R3-R5$	$R1-R5$	
	$R2-R3$	
	$R2-R4$	
	$R3-R4$	
	$R4-R5$	
	$R4-R6$	

5. Donner le coût d'une liaison fibre optique, le coût d'une liaison fast-ethernet et le coût d'une liaison ethernet. 0,25 pt

Fibre optique : coût = $\frac{10^8}{10^9} = 10^{-1} = 0{,}1$

Fast-ethernet : coût = $\frac{10^8}{100\times10^6} = 1$

Ethernet : coût = $\frac{10^8}{10\times10^6} = 10$

6. Donner la table de routage de $R1$, après la convergence, en indiquant pour chaque route : 0,5 pt

- Le nom du routeur destination ;
- Le nom du routeur passerelle éventuel ;
- Le coût total pour atteindre le routeur destination.

Destination	Passerelle	Coût
R1		0
R2	R2	1
R3	R3	0,1
R4	R3	1,1
R5	R3	0,2
R6	R3	2,1

7. Le routeur $R3$ tombe en panne. Donner la table de routage de $R1$, après la convergence, sous le même format qu'à la question précédente. 0,5 pt

Destination	Passerelle	Coût
R1		0
R2	R2	1
R3		infini
R4	R2 ou R5	2
R5	R5	1
R6	R2 ou R5	3

Partie B

On implémente un réseau sous Python à l'aide de la classe **`Reseau`** ci-après :

```
1   class Reseau:
2       def __init__(self):
3           self.adjacence = { }  # dictionnaire d'adjacence du réseau
4           self.tables = { }     # dictionnaire des tables de routage de tous les routeurs
5
6       def ajouteRouteur(self, routeur):
7           ''' ajoute routeur au dictionnaire d'adjacence, s'il n'existe pas déjà'''
8           ... (plusieurs lignes)
9
10      def ajouteLiaison(self,routeur1, routeur2):
11          ''' ajoute les liens entre les  routeurs 1 et 2 au dictionnaire d'adjacence'''
12          ... (plusieurs lignes)
13
14      def ajouteTableInitiale(self, routeur):
15          '''ajoute la table de routage initiale de routeur au dictionnaire des tables'''
16          ... (plusieurs lignes)
17
18      def ajouteTablesInitiales(self):
19          for routeur in self.adjacence:
20              self.ajouteTableInitiale(routeur)
```

Le code complémentaire ci-après crée un réseau nommé **`lycee`**, puis ajoute la liaison $R1 - R2$ (et donc les routeurs $R1$ et $R2$) et enfin ajoute le routeur $R3$ (sans liaison donc). Le dictionnaire d'adjacence est ensuite affiché, le résultat est mis en commentaire en ligne 32.

```
28  lycee = Reseau()
29
30  lycee.ajouteLiaison('R1', 'R2')
31  lycee.ajouteRouteur('R3')
32  print(lycee.adjacence)          # {'R1': ['R2'], 'R2': ['R1'], 'R3': []}
```

8. Donner les instructions en Python nécessaires à la création du réseau présenté en début d'exercice (cf : schéma 1). 0,5 pt

```
lycee.ajouteLiaison('R1', 'R2');  lycee.ajouteLiaison('R1', 'R3');
lycee.ajouteLiaison('R1', 'R5');
lycee.ajouteLiaison('R2', 'R3');  lycee.ajouteLiaison('R2', 'R4');
lycee.ajouteLiaison('R2', 'R6');
lycee.ajouteLiaison('R3', 'R4');  lycee.ajouteLiaison('R3', 'R5');
lycee.ajouteLiaison('R4', 'R5');  lycee.ajouteLiaison('R4', 'R6');
```

9. Recopier et compléter la méthode **`ajouteRouteur`** de la classe **`Reseau`** qui ajoute **`routeur`**, s'il n'existe pas déjà, au dictionnaire d'adjacence du réseau. 0,5 pt

```
def ajouteRouteur(self, routeur):
        ''' ajoute routeur au dictionnaire d'adjacence, s'il n'existe pas déjà'''
        if routeur not in self.adjacence :
            self.adjacence[routeur] = []
```

10. Recopier et compléter la méthode **`ajouteLiaison`** de la classe **`Reseau`** qui ajoute la liaison $R1 - R2$ (et donc aussi la liaison $R2 - R1$) au dictionnaire d'adjacence du réseau. 0,75 pt

```
def ajouteLiaison(self,routeur1, routeur2):
        ''' ajoute les liens entre les  routeurs 1 et 2 au dictionnaire d'adjacence'''
        self.ajouteRouteur(routeur1)
        self.ajouteRouteur(routeur2)
        self.adjacence[routeur1].append(routeur2)
        self.adjacence[routeur2].append(routeur1)
```

La table de routage initiale d'un routeur est construite directement à partir du dictionnaire d'adjacence du réseau, sans exploration du graphe, et contient une route vers chaque voisin, avec pour chaque route :

- Le nom du routeur destination ;
- Le nom du routeur passerelle éventuel ;
- Le nombre de sauts pour atteindre le routeur destination.

Le code complémentaire ci-après construit les tables initiales de tous les routeurs du réseau avec la méthode **`ajouteTablesInitiales`** (avec un « s » !). Le dictionnaire des tables est ensuite affiché. Le résultat attendu est mis en commentaire en lignes 35 à 37.

```
34  lycee.ajouteTablesInitiales()
35  print(lycee.tables)   # Affichage obtenu : {'R1': ['R1', None, 0, ['R2', 'R2', 1]],
36                        #                     'R2': ['R2', None, 0, ['R1', 'R1', 1]],
37                        #                     'R3': ['R3', None, 0]}
```

11. Recopier et compléter la méthode **`ajouteTableInitiale`** de la classe **`Reseau`** qui construit et ajoute la table initiale de **`routeur`** au dictionnaire des tables de routage, attribut de la classe **`Reseau`.** 1,25 pt

```
def ajouteTableInitiale(self, routeur):
        '''ajoute la table de routage initiale de routeur au dictionnaire des tables'''
        table = [routeur, None, 0]
        for voisin in self.adjacence[routeur]:
            table.append([voisin, voisin, 1])
        self.tables[routeur] = table
```