BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

Mars 2025 – Bac Blanc

N.S.I.
Numérique et Sciences Informatiques

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice n'est pas autorisé

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.
Ce sujet comporte 14 pages numérotées de 1 à 14

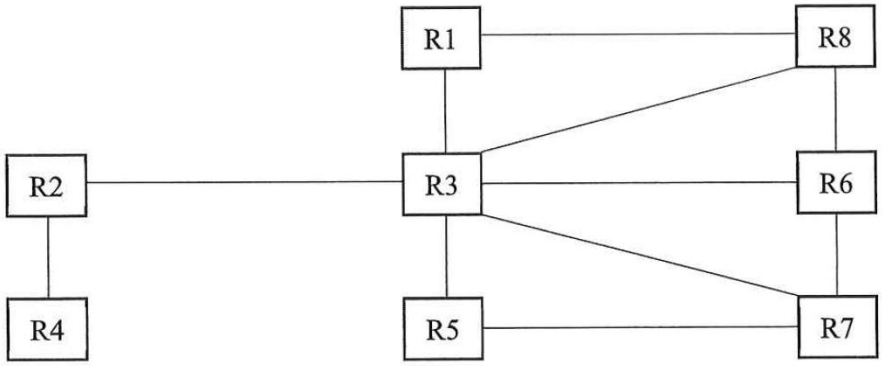
Exercice 1 : *(7 points)*

Les deux parties sont indépendantes.

Partie A

La responsable informatique doit gérer le réseau informatique de son entreprise représenté ci-dessous dans lequel R1, R2, R3, R4, R5, R6, R7 et R8 sont des routeurs. Elle décide d'utiliser le protocole RIP pour configurer les tables de routages.

Ce protocole est un protocole à vecteurs de distance. La métrique permettant de décider du meilleur chemin vers un routeur distant est le nombre de sauts.



1. La table de routage de R1 débute ainsi :

Destination	Passerelle	Métrique
R1	R1	0
R2	R3	2
R3	R3	1

Recopier puis compléter la table de routage de R1 selon le protocole RIP.

On donne le constructeur de la classe `Routeur`

```
class Routeur :
    def __init__(self, name) :
        self.nom = name
        self.table_routage = { self : (self, 0) }
        # le routeur est lié à lui-même (self)
```

L'attribut `table_routage` est un dictionnaire dont les clés sont des objets de type `Routeur` et la valeur est le couple `(pasr, m)` où `pasr` est un objet de type `Routeur` et `m` est la métrique selon le protocole RIP entre le routeur de la clé et le routeur `pasr`.

2. La méthode `ajout_destination(self, dest, pasr, m)` de la classe `Routeur`, prend en paramètres un routeur de destination `dest`, un routeur passerelle `pasr` et un entier `m`. Elle ajoute à la table de routage de `self` le routeur `dest` associé à la passerelle `pasr` et à la métrique `m`.

   Par exemple, on crée les routeurs R1, R2 et R3 puis on ajoute les destinations R2 et R3 à la table de routage de R1 à l'aide des commandes suivantes :

```
r1 = Routeur("R1")
r2 = Routeur("R2")
r3 = Routeur("R3")
r1.ajout_destination(r2, r3, 2)
r1.ajout_destination(r3, r3, 1)
```

   On dispose d'une méthode `afficher(self)` qui permet d'obtenir la table de routage du routeur `self`.

   Par exemple, la commande `r1.afficher()` affiche :

```
  ROUTEUR R1 : table de routage
dest.     | pass.    | metrique
R1        | R1       | 0
R2        | R3       | 2
R3        | R3       | 1
```

   a. Proposer les commandes nécessaires permettant d'ajouter les routeurs R4, R5 à la table de routage de R1.

   b. Écrire la méthode `ajout_destination(self, dest, pasr, m)`.

3. Écrire la méthode `voisins(self)` qui renvoie la liste des routeurs directement connectés au routeur `self`.

   Par exemple, `r1.voisins()` renverra la liste `[r3, r8]`, où `r3` et `r8` sont des objets de types `Routeur`.

4. La méthode `calcul_route(self, dest)` de la classe `Routeur` prend en paramètre un routeur de destination `dest` et renvoie la liste des routeurs parcourus lors d'une communication entre les routeurs `self` et `dest`.

Par exemple, `r1.calcul_route(r7)` renverra la liste `[r1, r3, r7]` où `r1`, `r3` et `r7` sont des objets de types `Routeur`.

Recopier et compléter le code de la méthode `calcul_route(self, dest)` ci-dessous

```
def calcul_route(self, dest) :
    route = [self]
    routeur_courant = route[-1]
    while ... :

        ...          # plusieurs lignes

    return route
```

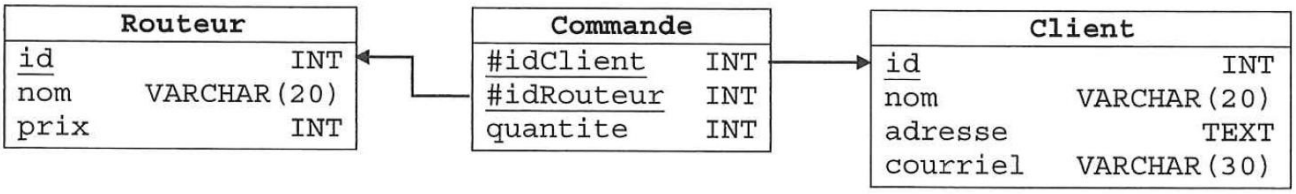
Partie B

Dans cette partie, on pourra utiliser les mots clés suivants du langage SQL.

`SELECT, INSERT INTO, WHERE, UPDATE, JOIN, ORDER BY`

La commande `ORDER BY propriété` permet de trier dans l'ordre croissant les résultats d'une requête selon l'attribut `propriété`.

Pour gérer son réseau informatique, la responsable achète son matériel à la société BeauReseau qui dispose d'une base de données dont le schéma relationnel est ci-dessous.



Les clés primaires sont soulignées et les clés étrangères sont précédées du caractère #.
Ainsi l'attribut `idRouteur` de la relation `Commande` est une clé étrangère qui fait référence à l'attribut `id` de la relation `Routeur` et l'attribut `idClient` de la relation `Commande` est une clé étrangère qui fait référence à l'attribut `id` de la relation `Client`.

Pour les questions où on demande des résultats de requêtes, on considèrera les extraits des tables remplies ainsi :

Table Routeur

id	nom	prix
1	C6Po-1000	1200
2	Coq6-300	6000
3	Al-200	6000
4	C6Po-9000	9000

Table Client

id	nom	adresse	courriel
1	Knuth	rue Donald, Tampa	dknuth@usa.org
2	Hooper	rue Grace, Boston	ghooper@usa.org
3	Torvalds	rue Linus, Helsinki	ltorvalds@finland.org
4	Pouzin	rue Louis, Paris	lpouzin@france.fr

Table Commande

idClient	idRouteur	quantite
1	1	4
1	3	1
2	1	1
2	1	5
3	2	3
4	4	1
4	1	5

5. On considère la requête d'insertion suivante

```
INSERT INTO Routeur(id, nom, prix) VALUES(3, 'Dali-32', 4000)
```

Expliquer pourquoi cette requête renvoie une erreur.

6. Proposer une requête qui renvoie le nom de tous les routeurs dont le prix est compris entre 2500 (inclus) et 7000 euro (inclus).

7. On considère la requête suivante :

```
SELECT nom FROM Client
JOIN Commande ON Commande.idClient = Client.id
WHERE Commande.quantite = 1
```

En considérant les extraits des tables fournies, préciser ce que renvoie cette requête.

8. Proposer une requête qui renvoie les noms triés dans l'ordre croissant des routeurs achetés par le client n°2.

Exercice 2 : *(6 points)*

Cet exercice porte sur les structures de données FILE et PILE, les graphes et les algorithmes de parcours.

Partie A

Une agence de voyages organise différentes excursions dans une région de France et propose la visite de certaines villes. Ces excursions peuvent être visualisées sur le graphe ci-dessous : les sommets désignent les villes, les arêtes représentent les routes pouvant être empruntées pour relier deux villes et les poids des arêtes représentent des distances, exprimées en kilomètre.

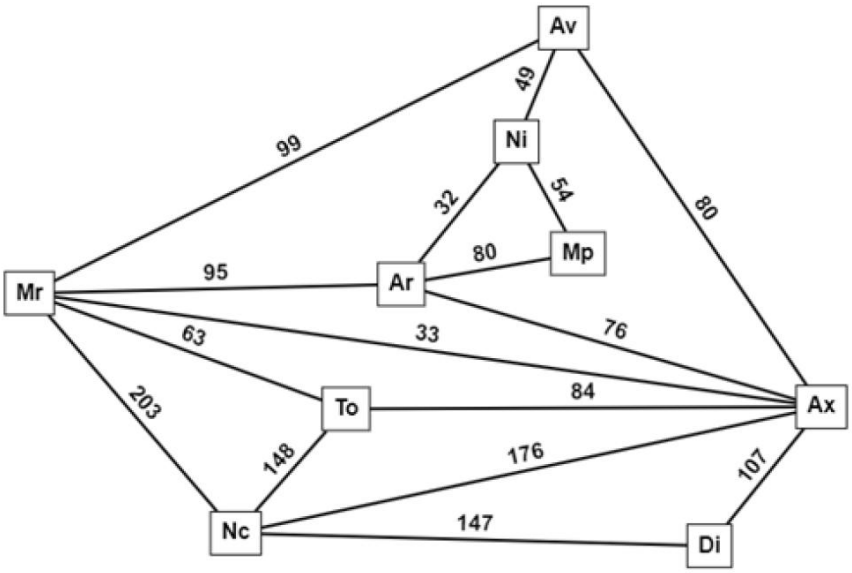


Figure 1. Graphe pondéré

1. Déterminer le plus court chemin allant du sommet Mp au sommet Nc et préciser la longueur, en kilomètres, de ce chemin. Aucune justification n'est attendue.

On souhaite toujours se rendre du sommet Mp au sommet Nc mais en visitant le minimum de villes.

2. Déterminer les deux chemins possibles.

Partie B

L'agence souhaite proposer un itinéraire permettant de visiter toutes les villes. On appelle G le graphe non pondéré ci-dessous.

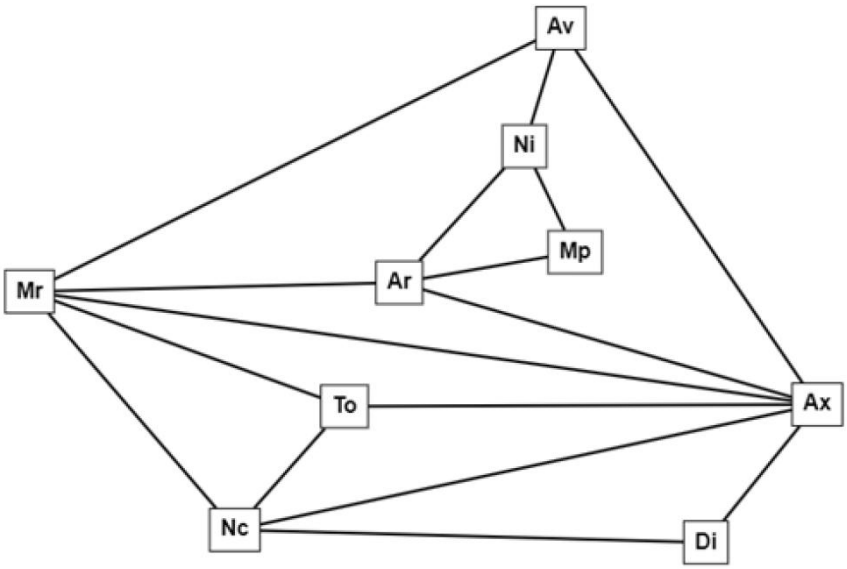


Figure 2. Graphe non pondéré

On choisit d'implémenter un graphe par listes d'adjacence, à l'aide d'un dictionnaire, en langage Python, dont :

- les clés sont les sommets du graphe ;
- la valeur associée à une clé est la liste des voisins de ce sommet clé.

Les sommets sont de type `str`.

3. Donner l'implémentation, en langage Python, du graphe de la figure 2. Le dictionnaire obtenu sera stocké dans une variable nommée `G`. Afin de faciliter la notation manuscrite ainsi que la lisibilité, écrire chaque couple clé/valeur sur une nouvelle ligne.

On considère une file.

4. Indiquer la signification des lettres dans les acronymes LIFO et FIFO.

5. Indiquer l'acronyme utilisé pour désigner la structure de file.

Voici, en langage Python, les opérations pouvant être effectuées sur une telle file :

- `creerFile()` : renvoie une file vide ;
- `estVide(F)` : renvoie `True` si la file `F` est vide et `False` sinon ;
- `enfiler(F, e)` : ajoute l'élément `e` dans la file `F` ;
- `defiler(F)` : renvoie l'élément à la tête de la file `F` en le retirant de la file `F`.

On donne la fonction `parcours` ci-dessous. Cette fonction prend en paramètres un dictionnaire `graphe` représentant un graphe sous la forme de listes d'adjacence, et une chaîne de caractères `sommet` représentant un sommet du graphe.

```
1  def parcours(graphe, sommet):
2      f = creerFile()
3      enfiler(f, sommet)
4      visite = [sommet]
5      while not estVide(f):
6          s = defiler(f)
7          for v in graphe[s]:
8              if not (v in visite):
9                  visite.append(v)
10                 enfiler(f, v)
11     return visite
```

6. Donner le résultat renvoyé par l'appel `parcours(G, 'Av')`.

7. Recopier, parmi les deux propositions ci-dessous, celle qui correspond au type de parcours de graphe réalisé par la fonction `parcours` :

- **proposition A :** parcours en largeur ;
- **proposition B :** parcours en profondeur.

Dans la suite de l'exercice, la distance entre deux sommets désignera le nombre d'arêtes séparant ces deux sommets. Ainsi définie, la distance entre les sommets Mp et Nc du graphe de la figure 2 est 3.

8. En modifiant la fonction `parcours`, écrire une fonction `distance` ayant pour paramètres `graphe`, un dictionnaire représentant un graphe sous la forme de listes d'adjacence, et une chaîne de caractères `sommet` représentant un sommet du graphe. Cette fonction renvoie un dictionnaire tel que :
   - les clés sont les sommets du graphe ;
   - la valeur associée à une clé est la distance entre ce sommet clé et le sommet d'origine `sommet`.
9. Donner le résultat renvoyé par l'appel `distance(G, 'Av')`.

On considère une pile.
Voici, en langage Python, les opérations pouvant être effectuées sur une telle pile :

- `creerPile()` : renvoie une pile vide ;
- `estVide(P)` : renvoie `True` si la pile `P` est vide et `False` sinon ;
- `empiler(P, e)` : ajoute l'élément `e` au sommet de la pile `P` ;
- `depiler(P)` : renvoie le sommet de la pile `P` en le retirant de la pile `P`

On donne, ci-dessous, le pseudo-code d'un algorithme de parcours d'un graphe `G` à partir d'un sommet `s` :

```
1  créer une pile p
2  empiler s dans p
3  créer une liste visite vide
4  tant que p n'est pas vide
5      x = depiler p
6      si x n'est pas dans la liste visite
7          ajouter x à la liste visite
8          pour chaque voisin v de x
9              empiler v dans p
10         fin pour
11     fin si
12 fin tant que
13 renvoyer visite
```

10. Traduire, dans le corps d'une fonction Python nommée `parcours2`, l'algorithme en pseudo-code donné précédemment. Cette fonction prendra pour paramètres `G`, un dictionnaire représentant un graphe sous la forme de listes d'adjacence, et une chaîne de caractères `s` représentant un sommet du graphe.

11. Donner un résultat possible renvoyé par l'appel `parcours2(G, 'Av')`.

Exercice 3 : *(7 points)*

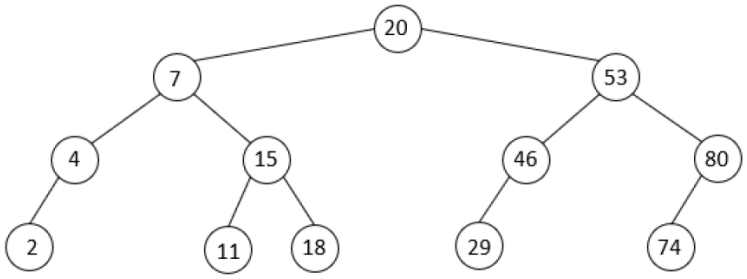
Cet exercice traite des arbres et de l'algorithmique.

Dans cet exercice, la taille d'un arbre est égale au nombre de ses nœuds et on convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1.

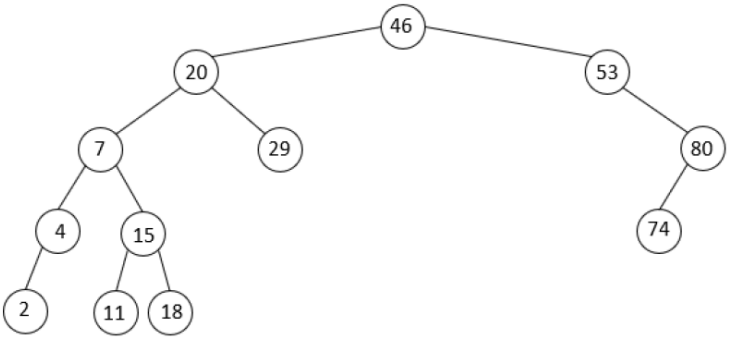
On utilisera la définition suivante : un arbre binaire de recherche est un arbre binaire, dans lequel

- on peut comparer les valeurs des nœuds : ce sont par exemple des nombres entiers, ou des lettres de l'alphabet ;
- si `x` est un nœud de cet arbre et `y` est un nœud du sous-arbre gauche de x, alors il faut que `y.valeur < x.valeur`
- si `x` est un nœud de cet arbre et `y` est un nœud du sous-arbre droit de `x`, alors il faut que `y.valeur ≥ x.valeur`

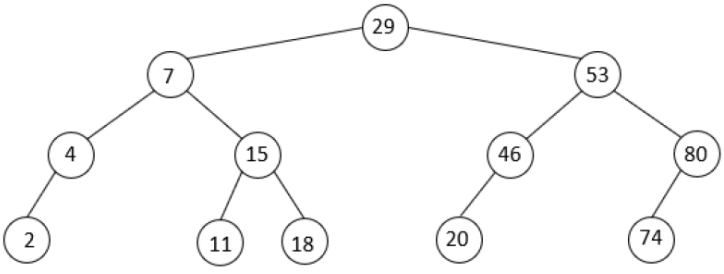
1. Parmi les trois arbres dessinés ci-dessous, recopier sur la copie le numéro correspondant à celui qui n'est pas un arbre binaire de recherche. Justifier.



Arbre 1



Arbre 2



Arbre 3

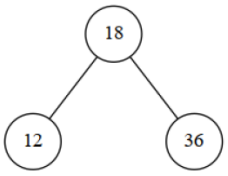
Une classe `ABR`, qui implémente une structure d'arbre binaire de recherche, possède l'interface suivante :

Numéro de lignes	Classe ABR
1	`class ABR :`
2	`    def __init__(self, valeur, sa_gauche, sa_droit):`
3	`        self.valeur = valeur #valeur de la racine`
4	`        self.sa_gauche = sa_gauche #sous-arbre gauche`
5	`        self.sa_droit = sa_droit #sous-arbre droit`
6	`    def inserer_noeud(self, valeur):`
7	`        """Renvoie un nouvel ABR avec le nœud de valeur 'valeur'`
8	`        inséré comme nouvelle feuille à sa position correcte"""`
9	`        # code non étudié dans cet exercice`
…	

On prendra la valeur `None` pour représenter un sous-arbre vide.

2. La construction d'un ABR se fait en insérant progressivement les valeurs à partir de la racine : la méthode `inserer_noeud` (dont le code n'est pas étudié dans cet exercice) place ainsi un nœud à sa "bonne place" comme feuille dans la structure, sans modifier le reste de la structure. On admet que la position de cette feuille est unique.

   a. En utilisant les méthodes de la classe ABR :
   - écrire l'instruction Python qui permet d'instancier un objet `a0`, de type ABR, ayant un seul nœud (la racine) de valeur 18.
   - écrire une séquence d'instructions qui permet ensuite d'insérer dans l'objet `a0` les deux feuilles de l'arbre de valeurs 12 et 36.

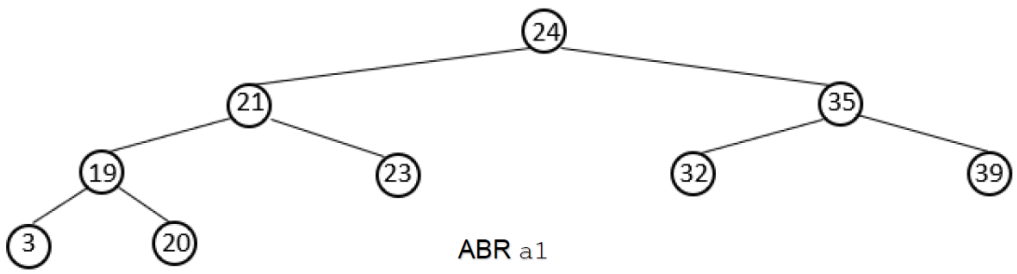


ABR `a0`

Selon l'ordre dans lequel les valeurs sont insérées, on construit des ABR ayant des structures différentes.

Voilà par exemple ci-dessous un ABR (nommé `a1`) obtenu en créant une instance de type ABR ayant un seul nœud (la racine) de valeur 24 puis en insérant successivement les valeurs dans l'ordre suivant :

21 ; 35 ; 19 ; 23 ; 32 ; 39 ; 3 ; 20



ABR `a1`

b. Dessiner sur la copie l'ABR (nommé `a2`) que l'on obtiendrait en créant une instance de type ABR ayant un seul nœud (la racine) de valeur 3 puis en insérant successivement les valeurs dans l'ordre suivant :

20 ; 19 ; 21 ; 23 ; 32 ; 24 ; 35 ; 39

c. Donner la hauteur des ABR `a1` et `a2`.

d. On complète la classe ABR avec une méthode `calculer_hauteur` qui renvoie la hauteur de l'arbre.
Recopier sur la copie les lignes 10 et 13 en les complétant par des commentaires et la ligne 14 en la complétant par une instruction dans le code ci-après de cette méthode.
On pourra utiliser la fonction Python `max` qui prend en paramètres deux nombres et renvoie le maximun de ces deux nombres.

Numéro de lignes	Méthode `calculer_hauteur`
1	`def calculer_hauteur(self):`
2	`    """ Renvoie la hauteur de l'arbre"""`
3	`    if self.sa_droit is None and self.sa_gauche is None:`
4	`    #l'arbre est réduit à une feuille`
5	`        return 1`
6	`    elif self.sa_droit is None`
7	`    #arbre avec une racine et seulement un sous-arbre gauche`
8	`        return 1 + self.sa_gauche.calculer_hauteur()`
9	`    elif self.sa_gauche is None:`
10	`    # à compléter`
11	`         return 1 + self.sa_droit.calculer_hauteur()`
12	`    else:`
13	`    # à compléter`
14	`         return à compléter`

3. La différence de hauteur entre l'ABR `a1` et l'ABR `a2` aura des conséquences lors de la recherche d'une valeur dans l'ABR.

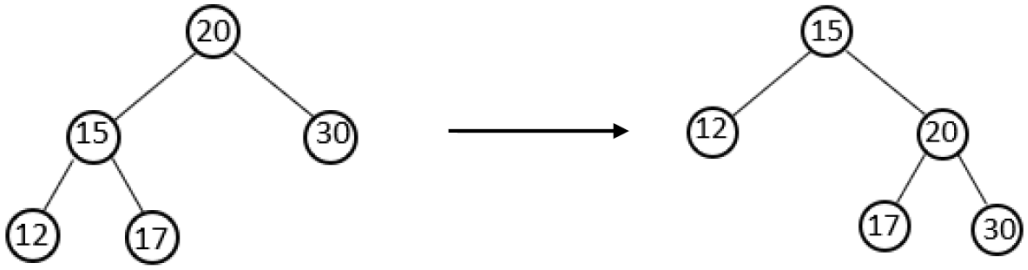
a. Recopier et compléter sur la copie les lignes 6, 8, 11 et 13 du code ci-dessous de la méthode `rechercher_valeur`, qui permet de tester la présence ou l'absence d'une valeur donnée dans l'ABR :

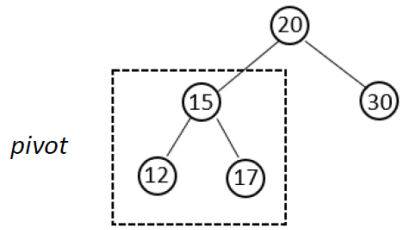
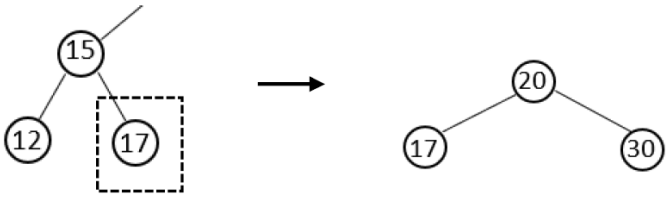
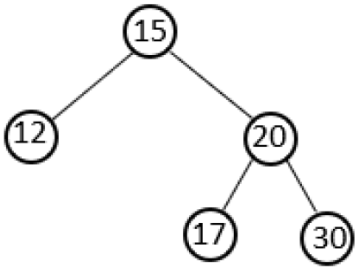
Numéro de lignes	Méthode `rechercher_valeur`
1	`def rechercher_valeur(self, v):`
2	`    """`
3	`    Renvoie True si la valeur v est trouvée dans l'ABR,`
4	`    False sinon`
5	`    """`
6	`    if à compléter`
7	`        return True`
8	`    elif à compléter and self.sa_gauche is not None:`
9	`        return self.sa_gauche.rechercher_valeur(v)`
10	`    elif v > self.valeur and self.sa_droit is not None:`
11	`        return à compléter`
12	`    else:`
13	`        return à compléter`

b. On admet que le nombre de fois où la méthode `rechercher_valeur` est appelée pour rechercher la valeur 39 dans l'ABR `a2` est 7.
Donner le nombre de fois où la méthode `rechercher_valeur` est appelée pour rechercher la valeur 20 dans l'ABR `a1`.

4. Il existe des algorithmes pour modifier la structure d'un ABR, afin par exemple de diminuer la hauteur d'un ABR ; on s'intéresse aux algorithmes appelés *rotation*, consistant à faire "pivoter" une partie de l'arbre autour d'un de ses nœuds.

L'exemple ci-dessous permet d'expliquer l'algorithme pour réaliser une rotation droite d'un ABR autour de sa racine :



On appelle *pivot* le sous-arbre gauche de la racine de l'arbre	
Le sous-arbre droit du pivot devient le sous-arbre gauche de la racine	
La racine ainsi modifiée devient le sous-arbre droit du pivot et la racine du pivot devient la nouvelle racine de l'ABR	

On admet que ces transformations conservent la propriété d'ABR de l'arbre.

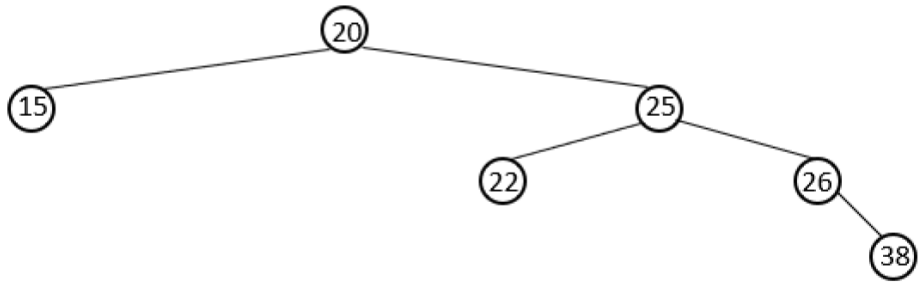
La méthode `rotation_droite` ci-après renvoie une nouvelle instance de type ABR, correspondant à une rotation droite de l'objet de type ABR à partir duquel elle est appelée :

Numéro de lignes	Méthode `rotation_droite`
1	`def rotation_droite(self):`
2	`    """ Renvoie une instance d'un ABR apres une rotation droite`
3	`        On suppose qu'il existe un sous-arbre gauche"""`
4	`    pivot = self.sa_gauche`
5	`    self.sa_gauche = pivot.sa_droit`
6	`    pivot.sa_droit = self`
7	`    return ABR(pivot.valeur,pivot.sa_gauche,pivot.sa_droit)`

Pour réaliser une rotation gauche, on suivra alors l'algorithme suivant :

- on appelle *pivot* le sous-arbre droit de la racine de l'arbre,
- le sous-arbre gauche du pivot devient le sous-arbre droit de la racine,
- la racine ainsi modifiée devient le sous-arbre gauche du pivot et la racine du pivot devient la nouvelle racine de l'ABR

a. En suivant les différentes étapes de cet algorithme, dessiner l'arbre obtenu après une rotation gauche de l'ABR suivant :



b. Écrire le code d'une méthode Python `rotation_gauche` qui réalise la rotation gauche d'un ABR autour de sa racine.