0.6 Les arbres

NSI terminale - JB Duthoit

EXERCICE 3 (4 points)

Cet exercice est consacré aux arbres binaires de recherche et à la notion d'objet.

1. Voici la définition d'une classe nommée `ArbreBinaire`, en Python :

Numéro de lignes	Classe `ArbreBinaire`
1	`class ArbreBinaire:`
2	`    """ Construit un arbre binaire """`
3	`    def __init__(self, valeur):`
4	`        """ Crée une instance correspondant`
5	`        à un état initial """`
6	`        self.valeur = valeur`
7	`        self.enfant_gauche = None`
8	`        self.enfant_droit = None`
9	`    def insert_gauche(self, valeur):`
10	`        """ Insère le paramètre valeur`
11	`        comme fils gauche """`
12	`        if self.enfant_gauche is None:`
13	`            self.enfant_gauche = ArbreBinaire(valeur)`
14	`        else:`
15	`            new_node = ArbreBinaire(valeur)`
16	`            new_node.enfant_gauche = self.enfant_gauche`
17	`            self.enfant_gauche = new_node`
18	`    def insert_droit(self, valeur):`
19	`        """ Insère le paramètre valeur`
20	`        comme fils droit """`
21	`        if self.enfant_droit is None:`
22	`            self.enfant_droit = ArbreBinaire(valeur)`
23	`        else:`
24	`            new_node = ArbreBinaire(valeur)`
25	`            new_node.enfant_droit = self.enfant_droit`
26	`            self.enfant_droit = new_node`
27	`    def get_valeur(self):`
28	`        """ Renvoie la valeur de la racine """`
29	`        return self.valeur`
30	`    def get_gauche(self):`
31	`        """ Renvoie le sous arbre gauche """`
32	`        return self.enfant_gauche`
33	`    def get_droit(self):`
34	`        """ Renvoie le sous arbre droit """`
35	`        return self.enfant_droit`

a. En utilisant la classe définie ci-dessus, donner un exemple d'attribut, puis un exemple de méthode.

b. Après avoir défini la classe `ArbreBinaire`, on exécute les instructions Python suivantes :

```
r = ArbreBinaire(15)
r.insert_gauche(6)
r.insert_droit(18)
a = r.get_valeur()
b = r.get_gauche()
c = b.get_valeur()
```

Donner les valeurs associées aux variables `a` et `c` après l'exécution de ce code.

On utilise maintenant la classe `ArbreBinaire` pour implémenter un arbre binaire de recherche.

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`.

2. On exécute le code Python suivant. Représenter graphiquement l'arbre ainsi obtenu.

```
racine_r = ArbreBinaire(15)

racine_r.insert_gauche(6)
racine_r.insert_droit(18)

r_6 = racine_r.get_gauche()
r_6.insert_gauche(3)
r_6.insert_droit(7)

r_18 = racine_r.get_droit()
r_18.insert_gauche(17)
r_18.insert_droit(20)

r_3 = r_6.get_gauche()
r_3.insert_gauche(2)
```

3. On a représenté sur la figure 1 ci-dessous un arbre. Justifier qu'il ne s'agit pas d'un arbre binaire de recherche. Redessiner cet arbre sur votre copie en conservant l'ensemble des valeurs {2,3,5,10,11,12,13} pour les nœuds afin qu'il devienne un arbre binaire de recherche.

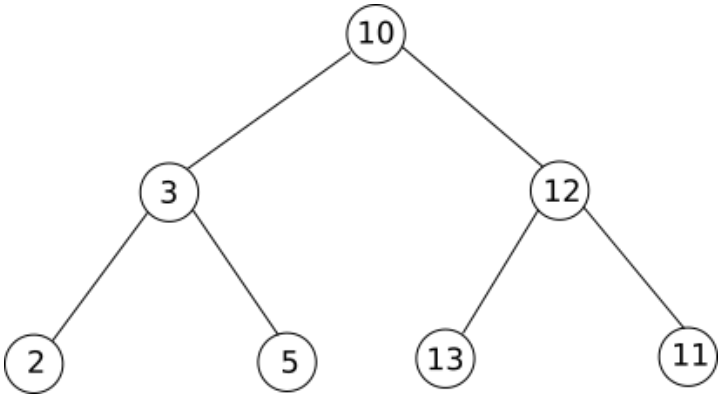


Figure 1

4. On considère qu'on a implémenté un objet `ArbreBinaire` nommé A représenté sur la figure 2.

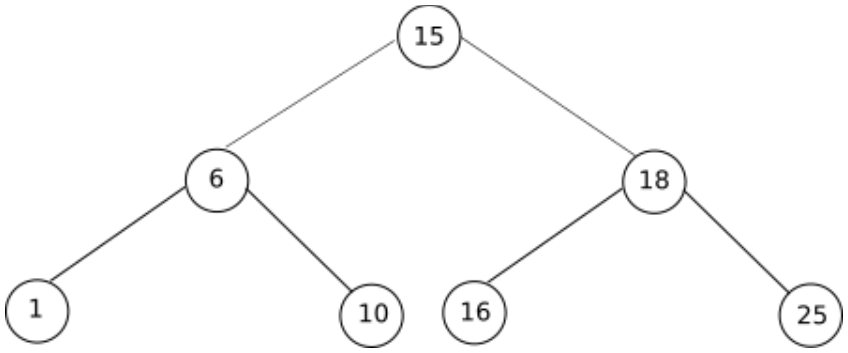


Figure 2

On définit la fonction `parcours_infixe` suivante, qui prend en paramètre un objet `ArbreBinaire T` et un second paramètre `parcours` de type liste.

Numéro de lignes	Fonction `parcours_infixe`
1	`def parcours_infixe(T, parcours):`
2	`    """ Affiche la liste des valeurs de l'arbre """`
3	`    if T is not None:`
4	`        parcours_infixe(T.get_gauche(), parcours)`
5	`        parcours.append(T.get_valeur())`
6	`        parcours_infixe(T.get_droit(), parcours)`
7	`    return parcours`

Donner la liste renvoyée par l'appel suivant : `parcours_infixe(A,[ ])`.

EXERCICE 3 (6 points)

L'exercice porte sur les arbres binaires de recherche et la programmation objet.

Dans un entrepôt de e-commerce, un robot mobile autonome exécute successivement les tâches qu'il reçoit tout au long de la journée.

La mémorisation et la gestion de ces tâches sont assurées par une structure de données.

1. Dans l'hypothèse où les tâches devraient être extraites de cette structure (pour être exécutées) dans le même ordre qu'elles ont été mémorisées, préciser si ce fonctionnement traduit le comportement d'une file ou d'une pile. Justifier.

En réalité, selon l'urgence des tâches à effectuer, on associe à chacune d'elles, lors de la mémorisation, un indice de priorité (nombre entier) distinct : il n'y a pas de valeur en double.

Plus cet indice est faible, plus la tâche doit être traitée prioritairement.

La structure de données retenue est assimilée à un arbre binaire de recherche (ABR) dans lequel chaque nœud correspond à une tâche caractérisée par son indice de priorité.

Rappel : Dans un arbre binaire de recherche, chaque nœud est caractérisé par une valeur (ici l'indice de priorité), telle que chaque nœud du sous-arbre gauche a une valeur strictement inférieure à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une valeur strictement supérieure à celle-ci.
Cette structure de données présente l'avantage de mettre efficacement en œuvre l'insertion ou la suppression de nœuds, ainsi que la recherche d'une valeur.

Par exemple, le robot a reçu successivement, dans l'ordre, des tâches d'indice de priorité 12, 6, 10, 14, 8 et 13. En partant d'un arbre binaire de recherche vide, l'insertion des différentes priorités dans cet arbre donne la figure 1.

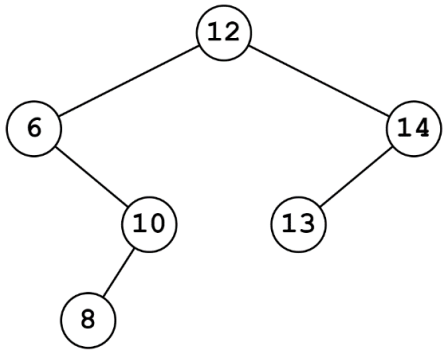


Figure 1 : Exemple d'un arbre binaire

2. En utilisant le vocabulaire couramment utilisé pour les arbres, préciser le terme qui correspond :

- **a.** au nombre de tâches restant à effectuer, c'est-à-dire le nombre total de nœuds de l'arbre ;
- **b.** au nœud représentant la tâche restant à effectuer la plus ancienne ;

c. au nœud représentant la dernière tâche mémorisée (la plus récente).

3. Lorsque le robot reçoit une nouvelle tâche, on déclare un nouvel objet, instance de la classe `Noeud`, puis on l'insère dans l'arbre binaire de recherche (instance de la classe `ABR`) du robot. Ces 2 classes sont définies comme suit :

```
 1  class Noeud:
 2      def __init__(self, tache, indice):
 3          self.tache = tache   #ce que doit accomplir le robot
 4          self.indice = indice #indice de priorité (int)
 5          self.gauche = ABR()  #sous-arbre gauche vide (ABR)
 6          self.droite = ABR()  #sous-arbre droit vide (ABR)
 7
 8
 9  class ABR:
10      #arbre binaire de recherche initialement vide
11      def __init__(self):
12          self.racine = None #arbre vide
13          #Remarque : si l'arbre n'est pas vide, racine est
14          #une instance de la classe Noeud
15
16      def est_vide(self):
17          """renvoie True si l'arbre autoréférencé est vide,
18          False sinon"""
19          return self.racine == None
20
21      def insere(self, nouveau_noeud):
22          """insere un nouveau noeud, instance de la classe
23          Noeud, dans l'ABR"""
24          if self.est_vide():
25              self.racine = nouveau_noeud
26          elif self.racine.indice …… nouveau_noeud.indice
27              self.racine.gauche.insere(nouveau_noeud)
28          else:
29              self.racine.droite.insere(nouveau_noeud)
```

a. Donner les noms des attributs de la classe `Noeud`.

b. Expliquer en quoi la méthode `insere` est dite récursive et justifier rapidement qu'elle se termine.

c. Indiquer le symbole de comparaison manquant dans le test à la **ligne 26** de la méthode `insere` pour que l'arbre binaire de recherche réponde bien à la définition de l'encadré « **Rappel** » de la page 6.

d. On considère le robot dont la liste des tâches est représentée par l'arbre de la figure 1. Ce robot reçoit, successivement et dans l'ordre, des tâches d'indice de priorité 11, 5, 16 et 7, sans avoir accompli la moindre tâche entretemps. Recopier et compléter la figure 1 après l'insertion de ces nouvelles tâches.

4. Avant d'insérer une nouvelle tâche dans l'arbre binaire de recherche, il faut s'assurer que son indice de priorité n'est pas déjà présent.

Écrire une méthode `est_present` de la classe `ABR` qui répond à la description :

```
41  def est_present(self, indice_recherche) :
42      """renvoie True si l'indice de priorité indice_recherche
43      (int) passé en paramètre est déjà l'indice d'un nœud
44      de l'arbre, False sinon"""
```

5. Comme le robot doit toujours traiter la tâche dont l'indice de priorité est le plus petit, on envisage un parcours infixe de l'arbre binaire de recherche.

a. Donner l'ordre des indices de priorité obtenus à l'aide d'un parcours infixe de l'arbre binaire de recherche de la **figure 1**.

b. Expliquer comment exploiter ce parcours pour déterminer la tâche prioritaire.

6. Afin de ne pas parcourir tout l'arbre, il est plus efficace de rechercher la tâche du nœud situé le plus à gauche de l'arbre binaire de recherche : il correspond à la tâche prioritaire.

Recopier et compléter la méthode récursive `tache_prioritaire` de la classe `ABR`:

```
61  def tache_prioritaire(self):
62      """renvoie la tache du noeud situé le plus
63      à gauche de l'ABR supposé non vide"""
64      if self.racine.…….est_vide():#pas de nœud plus à gauche
65          return self.racine.……
66      else:
67          return self.racine.gauche.……()
```

7. Une fois la tâche prioritaire effectuée, il est nécessaire de supprimer le nœud correspondant pour que le robot passe à la tâche suivante :

- Si le nœud correspondant à la tâche prioritaire est une feuille, alors il est simplement supprimé de l'arbre (cette feuille devient un arbre vide)
- Si le nœud correspondant à la tâche prioritaire a un sous-arbre droit non vide, alors ce sous-arbre droit remplace le nœud prioritaire qui est alors écrasé, même s'il s'agit de la racine.

Dessiner alors, pour chaque étape, l'arbre binaire de recherche (seuls les indices de priorités seront représentés) obtenu pour un robot, initialement sans tâche, et qui a, successivement dans l'ordre :

- étape 1 : reçu une tâche d'indice de priorité 14 à accomplir
- étape 2 : reçu une tâche d'indice de priorité 11 à accomplir
- étape 3 : reçu une tâche d'indice de priorité 8 à accomplir
- étape 4 : accompli sa tâche prioritaire
- étape 5 : reçu une tâche d'indice de priorité 12 à accomplir
- étape 6 : accompli sa tâche prioritaire
- étape 7 : accompli sa tâche prioritaire
- étape 8 : reçu une tâche d'indice de priorité 15 à accomplir

- étape 9 : reçu une tâche d'indice de priorité 19 à accomplir
- étape 10 : accompli sa tâche prioritaire

EXERCICE 5 (4 points)

Cet exercice traite du thème « algorithmique », et principalement des algorithmes sur les arbres binaires.

On manipule ici les arbres binaires avec trois fonctions :

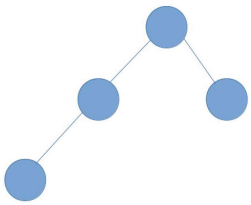
- `est_vide(A)` qui renvoie `True` si l'arbre binaire `A` est vide, `False` s'il ne l'est pas ;
- `sous_arbre_gauche(A)` qui renvoie le sous-arbre gauche de l'arbre binaire `A` ;
- `sous_arbre_droit(A)` qui renvoie le sous-arbre droit de l'arbre binaire `A`.

L'arbre binaire renvoyé par les fonctions `sous_arbre_gauche` et `sous_arbre_droit` peut éventuellement être l'arbre vide.

On définit la **hauteur** d'un arbre binaire non vide de la façon suivante :

- si ses sous-arbres gauche et droit sont vides, sa hauteur est 0 ;
- si l'un des deux au moins est non vide, alors sa hauteur est égale à 1 + M, où M est la plus grande des hauteurs de ses sous-arbres (gauche et droit) non vides.

1. a. Donner la hauteur de l'arbre ci-dessous.



b. Dessiner sur la copie un arbre binaire de hauteur 4.

La hauteur d'un arbre est calculée par l'algorithme récursif suivant :

```
1   Algorithme hauteur(A):
2     test d'assertion : A est supposé non vide
3     si sous_arbre_gauche(A) vide et sous_arbre_droit(A) vide:
4       renvoyer 0
5     sinon, si sous_arbre_gauche(A) vide:
6       renvoyer 1 + hauteur(sous_arbre_droit(A))
7     sinon, si ... :
8       renvoyer ...
9     sinon:
10      renvoyer 1 + max(hauteur(sous_arbre_gauche(A)),
11                       hauteur(sous_arbre_droit(A)))
```

2. Recopier sur la copie les lignes 7 et 8 en complétant les points de suspension.

3. On considère un arbre binaire R dont on note G le sous-arbre gauche et D le sous-arbre droit. On suppose que R est de hauteur 4 et G de hauteur 2.

   a. Justifier le fait que D n'est pas l'arbre vide et déterminer sa hauteur.

   b. Illustrer cette situation par un dessin.

Soit un arbre binaire non vide de hauteur h. On note n le nombre de nœuds de cet arbre. On admet que $h+1 \leqslant n \leqslant 2^{h+1}-1$.

4. a. Vérifier ces inégalités sur l'arbre binaire de la question 1.a.

   b. Expliquer comment construire un arbre binaire de hauteur h quelconque ayant h+1 nœuds.

   c. Expliquer comment construire un arbre binaire de hauteur h quelconque ayant $2^{h+1}-1$ nœuds.

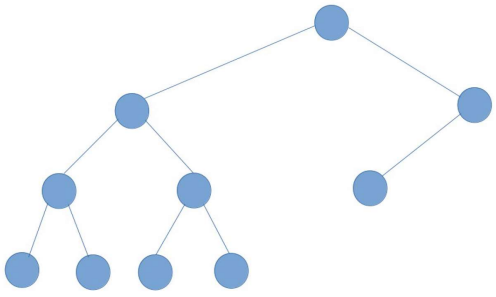
   Indication : $2^{h+1}-1 = 1+2+4+...+2^h$.

L'objectif de la fin de l'exercice est d'écrire le code d'une fonction `fabrique(h, n)` qui prend comme paramètres deux nombres entiers positifs `h` et `n` tels que $h+1 < n < 2^{h+1}-1$, et qui renvoie un arbre binaire de hauteur `h` à `n` nœuds.

Pour cela, on a besoin des deux fonctions suivantes:

- `arbre_vide()`, qui renvoie un arbre vide ;
- `arbre(gauche, droit)` qui renvoie l'arbre de fils gauche `gauche` et de fils droit `droit`.

5. Recopier sur la copie l'arbre binaire ci-dessous et numéroter ses nœuds de 1 en 1 en commençant à 1, en effectuant un parcours en profondeur préfixe.



La fonction `fabrique` ci-dessous a pour but de répondre au problème posé. Pour cela, la fonction `annexe` utilise la valeur de `n`, qu'elle peut modifier, et renvoie un arbre binaire de hauteur `hauteur_max` dont le nombre de nœuds est égal à la valeur de `n` au moment de son appel.

```
1.  def fabrique(h, n):
2.    def annexe(hauteur_max):
3.      if n == 0 :
4.        return arbre_vide()
5.      elif hauteur_max == 0:
6.        n = n - 1
7.        return ...
8.      else:
9.        n = n - 1
10.       gauche = annexe(hauteur_max - 1)
11.       droite = ...
12.       return arbre(gauche, droite)
13.   return annexe(h)
```

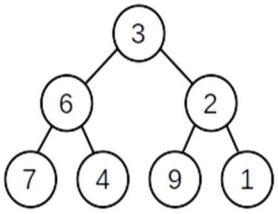
6. Recopier sur la copie les lignes 7 et 11 en complétant les points de suspension.

EXERCICE 4 (4 points)

Cet exercice, composé de deux parties A et B, porte sur le parcours des arbres binaires, le principe "diviser pour régner" et la récursivité.

Cet exercice traite du calcul de la somme d'un arbre binaire. Cette somme consiste à additionner toutes les valeurs numériques contenues dans les nœuds de l'arbre.

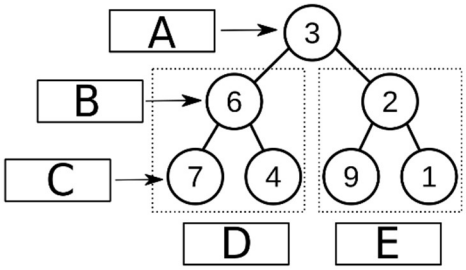
L'arbre utilisé *dans les parties A et B est le suivant :*



Partie A : Parcours d'un arbre

1. Donner la somme de l'arbre précédent. Justifier la réponse en explicitant le calcul qui a permis de l'obtenir.

2. Indiquer la lettre correspondante aux noms 'racine', 'feuille', 'nœud', 'SAG' (Sous Arbre Gauche) et 'SAD' (Sous Arbre Droit). Chaque lettre **A**, **B**, **C**, **D** et **E** devra être utilisée une seule fois.



Arbre avec les lettres à associer

3. Parmi les quatre propositions A, B, C et D ci-dessous, donnant un parcours en largeur d'abord de l'arbre, une seule est correcte. Indiquer laquelle.
Proposition A : 7 - 6 - 4 - 3 - 9 - 2 - 1
Proposition B : 3 - 6 - 7 - 4 - 2 - 9 - 1
Proposition C : 3 - 6 - 2 - 7 - 4 - 9 - 1
Proposition D : 7 - 4 - 6 - 9 - 1 - 2 – 3

4. Écrire en langage Python la fonction `somme` qui prend en paramètre une liste de nombres et qui renvoie la somme de ses éléments.
Exemple : `somme([1, 2, 3, 4])` est égale à 10.

5. La fonction `parcourir(arbre)` pourrait se traduire en langage naturel par :

```
parcourir(A):
    L = liste_vide
    F = file_vide
    enfiler A dans F
    Tant que F n'est pas vide
        défiler S de F
        ajouter la valeur de la racine de S dans L
        Pour chaque sous arbre SA non vide de S
            enfiler SA dans F
    renvoyer L
```

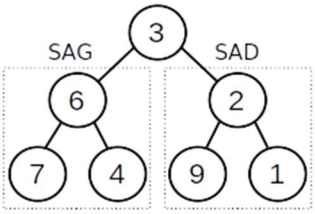
Donner le type de parcours obtenu grâce à la fonction `parcourir`.

Partie B : Méthode 'diviser pour régner'

6. Parmi les quatre propositions A,B, C et D ci-dessous, indiquer la seule proposition correcte.

En informatique, le principe diviser pour régner signifie :
Proposition A : diviser une fonction en deux fonctions plus petites
Proposition B : utiliser plusieurs modules
Proposition C : séparer les informations en fonction de leur types
Proposition D : diviser un problème en deux problèmes plus petits et indépendants.

7. L'arbre présenté dans le problème peut être décomposé en racine et sous arbres :



Indiquer dans l'esprit de 'diviser pour régner' l'égalité donnant la somme d'un arbre en fonction de la somme des sous arbres et de la valeur numérique de la racine.

8. Écrire en langage Python une fonction récursive `calcul_somme(arbre)`. Cette fonction calcule la somme de l'arbre passé en paramètre.

Les fonctions suivantes sont disponibles :

- `est_vide(arbre)` : renvoie `True` si `arbre` est vide et renvoie `False` sinon ;
- `valeur_racine(arbre)` : renvoie la valeur numérique de la racine de `arbre` ;
- `arbre_gauche(arbre)` : renvoie le sous arbre gauche de `arbre` ;
- `arbre_droit(arbre)` : renvoie le sous arbre droit de `arbre`.

EXERCICE 1 (4 points)

Cet exercice porte sur les arbres binaires de recherche, la programmation orientée objet et la récursivité.

Dans cet exercice, la taille d'un arbre est le nombre de nœuds qu'il contient. Sa hauteur est le nombre de nœuds du plus long chemin qui joint le nœud racine à l'une des feuilles (nœuds sans sous-arbres). On convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1 et la hauteur de l'arbre vide vaut 0.

1. On considère l'arbre binaire représenté ci-dessous:

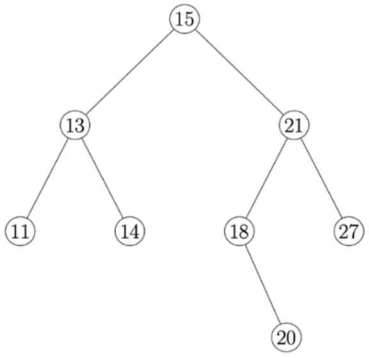


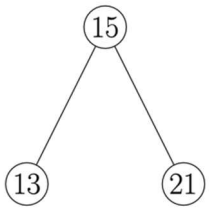
Figure 1

a. Donner la taille de cet arbre.
b. Donner la hauteur de cet arbre.
c. Représenter sur la copie le sous-arbre droit du nœud de valeur 15.
d. Justifier que l'arbre de la figure 1 est un arbre binaire de recherche.
e. On insère la valeur 17 dans l'arbre de la figure 1 de telle sorte que 17 soit une nouvelle feuille de l'arbre et que le nouvel arbre obtenu soit encore un arbre binaire de recherche. Représenter sur la copie ce nouvel arbre.

2. On considère la classe `Noeud` définie de la façon suivante en Python :

```
1  class Noeud:
2     def __init__(self, g, v, d):
3        self.gauche = g
4        self.valeur = v
5        self.droit = d
```

a. Parmi les trois instructions **(A)**, **(B)** et **(C)** suivantes, écrire sur la copie la lettre correspondant à celle qui construit et stocke dans la variable `abr` l'arbre représenté ci-contre.



(A) `abr=Noeud(Noeud(Noeud(None,13,None),15,None),21,None)`

(B) `abr=Noeud(None,13,Noeud(Noeud(None,15,None),21,None))`

(C) `abr=Noeud(Noeud(None,13,None),15,Noeud(None,21,None))`

b. Recopier et compléter la ligne 7 du code de la fonction `ins` ci-dessous qui prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et qui renvoie l'arbre obtenu suite à l'insertion de la valeur `v` dans l'arbre `abr`. Les lignes 8 et 9 permettent de ne pas insérer la valeur `v` si celle-ci est déjà présente dans `abr`.

```
1  def ins(v, abr):
2      if abr is None:
3          return Noeud(None, v, None)
4      if v > abr.valeur:
5          return Noeud(abr.gauche,abr.valeur,ins(v,abr.droit))
6      elif v < abr.valeur:
7          return ........................................
8      else:
9          return abr
```

3. La fonction `nb_sup` prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et renvoie le nombre de valeurs supérieures ou égales à la valeur `v` dans l'arbre `abr`.

Le code de cette fonction `nb_sup` est donné ci-dessous :

```
1  def nb_sup(v, abr):
2      if abr is None:
3          return 0
4      else:
5          if abr.valeur >= v:
6              return 1+nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
7          else:
8              return nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
```

a. On exécute l'instruction `nb_sup(16, abr)` dans laquelle `abr` est l'arbre initial de la figure 1. Déterminer le nombre d'appels à la fonction `nb_sup`.

b. L'arbre passé en paramètre étant un arbre binaire de recherche, on peut améliorer la fonction `nb_sup` précédente afin de réduire ce nombre d'appels. Écrire sur la copie le code modifié de cette fonction.

EXERCICE 2 (4 points)

Cet exercice porte sur les structures de données.

La poussette est un jeu de cartes en solitaire. Cet exercice propose une version simplifiée de ce jeu basée sur des nombres.

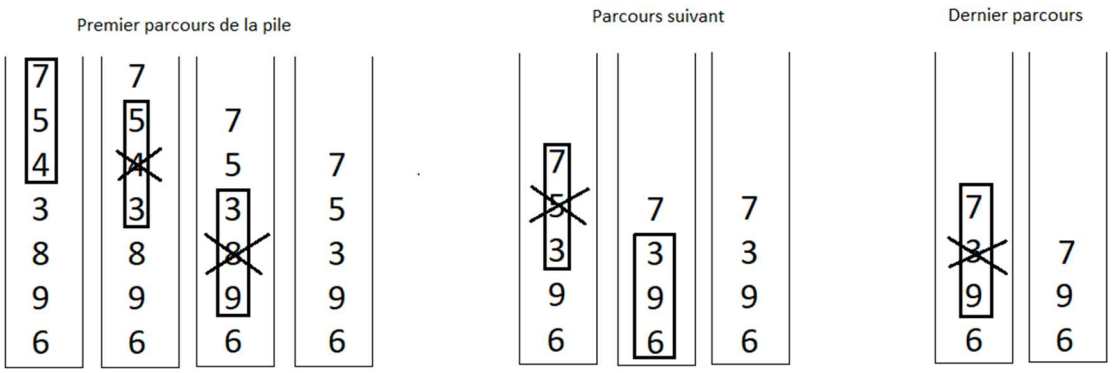
On considère une pile constituée de nombres entiers tirés aléatoirement. Le jeu consiste à réduire la pile suivant la règle suivante : quand la pile contient du haut vers le bas un triplet dont les termes du haut et du bas sont de même parité, on supprime l'élément central.

Par exemple :

- Si la pile contient du haut vers le bas, le triplet 1 0 3, on supprime le 0.
- Si la pile contient du haut vers le bas, le triplet 1 0 8, la pile reste inchangée.

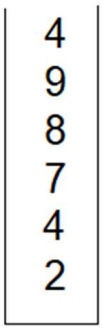
On parcourt la pile ainsi de haut en bas et on procède aux réductions. Arrivé en bas de la pile, on recommence la réduction en repartant du sommet de la pile jusqu'à ce que la pile ne soit plus réductible. Une partie est « gagnante » lorsque la pile finale est réduite à deux éléments exactement.

Voici un exemple détaillé de déroulement d'une partie.

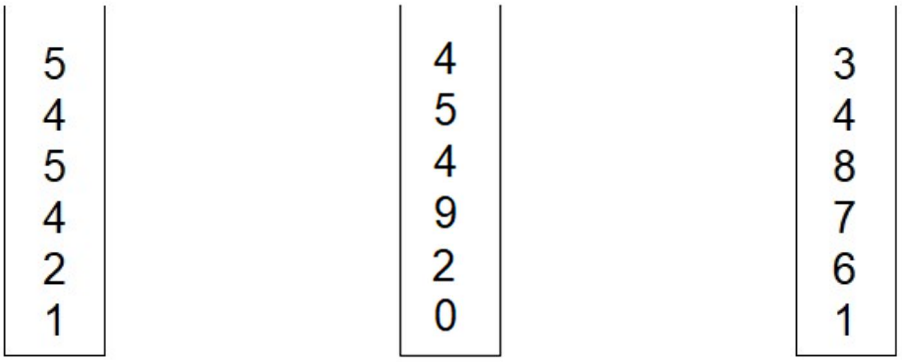


1.

a. Donner les différentes étapes de réduction de la pile suivante :



b. Parmi les piles proposées ci-dessous, donner celle qui est gagnante.



Pile A Pile B Pile C

L'interface d'une pile est proposée ci-dessous. On utilisera uniquement les fonctions figurant dans le tableau suivant :

Structure de données abstraite : Pile
• `creer_pile_vide()` renvoie une pile vide
• `est_vide(p)` renvoie `True` si `p` est vide, `False` sinon
• `empiler(p, element)` ajoute `element` au sommet de `p`
• `depiler(p)` retire l'élément au sommet de `p` et le renvoie
• `sommet(p)` renvoie l'élément au sommet de `p` sans le retirer de `p`
• `taille(p)` : renvoie le nombre d'éléments de `p`

2. La fonction `reduire_triplet_au_sommet` permet de supprimer l'élément central des trois premiers éléments en partant du haut de la pile, si l'élément du bas et du haut sont de même parité. Les éléments dépilés et non supprimés sont replacés dans le bon ordre dans la pile.
Recopier et compléter sur la copie le code de la fonction `reduire_triplet_au_sommet` prenant une pile `p` en paramètre et qui la modifie en place. Cette fonction ne renvoie donc rien.

```
1  def reduire_triplet_au_sommet(p):
2      a = depiler(p)
3      b = depiler(p)
4      c = sommet(p)
5      if a % 2 != .... :
6          empiler(p, ...)
7      empiler(p, ...)
```

3. On se propose maintenant d'écrire une fonction `parcourir_pile_en_reduisant` qui parcourt la pile du haut vers le bas en procédant aux réductions pour chaque triplet rencontré quand cela est possible.

a. Donner la taille minimale que doit avoir une pile pour être réductible.

b. Recopier et compléter sur la copie :

```
1  def parcourir_pile_en_reduisant(p):
2      q = creer_pile_vide()
3      while taille(p) >= ....:
4          reduire_triplet_au_sommet(p)
5          e = depiler(p)
6          empiler(q, e)
7      while not est_vide(q):
8          ............
9          ............
10      return p
```

4. Partant d'une pile d'entiers `p`, on propose ici d'implémenter une fonction récursive `jouer` renvoyant la pile `p` entièrement simplifiée. Une fois la pile parcourue de haut en bas et réduite, on procède à nouveau à sa réduction à condition que cela soit possible. Ainsi :

- Si la pile `p` n'a pas subi de réduction, on la renvoie.
- Sinon on appelle à nouveau la fonction `jouer` , prenant en paramètre la pile réduite.

Recopier et compléter sur la copie le code ci-dessous :

```
1  def jouer(p):
2      q = parcourir_pile_en_reduisant(p)
3      if ............. :
4          return p
5      else:
6          return jouer(...)
```

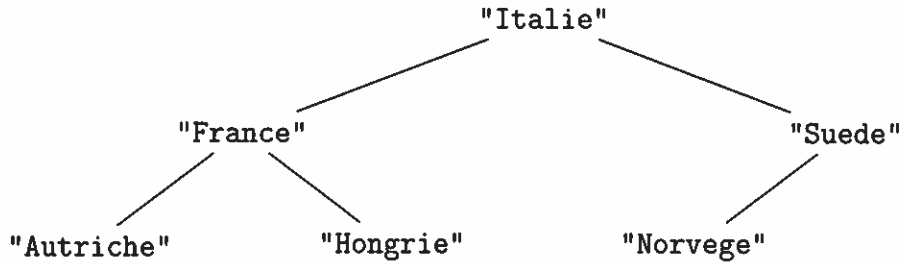
Exercice 3 (4 points).

Cet exercice porte sur les arbres binaires de recherche et leurs algorithmes associés.

Les arbres binaires de recherche considérés ici sont des arbres binaires où les nœuds désignent des chaînes de caractères et pour lesquelles la valeur de chaque nœud est supérieure à celles des nœuds de son enfant gauche, et inférieure à celles des nœuds de son enfant droit.
La relation d'ordre notée $<$ est ici la relation d'ordre alphabétique.

Dans cet exercice, on utilisera la convention suivante : la hauteur d'un arbre binaire ne comportant qu'un nœud est 1.
Dans cet exercice les arbres binaires de recherche ne contiennent que des noms de pays tous distincts.

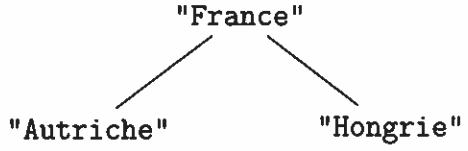
On considère l'arbre binaire de recherche suivant :



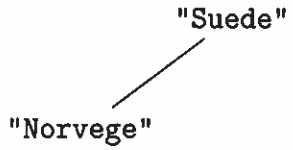
1. (a) Donner sans justification la hauteur de cet arbre.
   (b) Donner sans justification la valeur booléenne de l'expression `"Allemagne" < "Portugal"`.
   (c) Recopier l'arbre après l'ajout de `"Allemagne"`, de `"Portugal"` et de `"Luxembourg"` dans cet ordre.

Pour les questions 2, 3 et 4, on traite l'arbre initial, donc sans l'ajout de `"Allemagne"`, `"Portugal"` et `"Luxembourg"`.

2. On souhaite parcourir l'arbre. Indiquer l'ordre de visite des nœuds lors d'un parcours en largeur.
3. On souhaite écrire une fonction pour déterminer si le nom d'un pays est dans l'arbre.
   On dispose pour cela de :
   - la fonction `est_vide` qui prend en paramètre un arbre `arb`. Cette fonction renvoie `True` si l'arbre `arb` est vide, `False` sinon ;
   - la fonction `gauche` qui prend en paramètre un arbre `arb` et renvoie son sous-arbre gauche.
     Exemple : si `A` est notre arbre initial, `gauche(A)` renvoie



   - la fonction `droite`, qui prend en paramètre un arbre `arb` et renvoie son sous-arbre droit.
     Exemple : si `A` est notre arbre initial, `droite(A)` renvoie :



— la fonction `racine`, qui prend en paramètre un arbre `arb` et qui renvoie la valeur de la racine de l'arbre.

Exemple : `racine(A)` renvoie `"Italie"`.

Recopier, en complétant les lignes, 2, 6, 7 et 10, la fonction `recherche` donnée ci-dessous et écrite en Python. Cette fonction prend en paramètre un arbre `arb` et une valeur `val`. L'appel `recherche(arb, val)` renvoie un booléen (`True` si la valeur `val` est dans l'arbre `arb`, `False` sinon).

```
1  def recherche(arb, val):
2      """____________________"""
3      if est_vide(arb):
4          return False
5      if val == racine(arb):
6          return ________
7      if val _________________:
8          return recherche (gauche(arb), val)
9      else:
10         return ________
```

4. Écrire une fonction récursive `taille` permettant de déterminer le nombre de pays présent dans un arbre.

   Cette fonction prendra en paramètre un arbre `arb` et renverra un entier.

Exercice 1 (4 points).

Cet exercice porte sur les listes, les arbres binaires de recherche et la programmation orientée objet.

Lors d'une compétition de kayak, chaque concurrent doit descendre le même cours d'eau en passant dans des portes en un minimum de temps. Si le concurrent touche une porte, il se voit octroyé une pénalité en secondes. Son résultat final est le temps qu'il a mis pour descendre le cours d'eau auquel est ajouté l'ensemble des pénalités qu'il a subies.
Un gestionnaire de course de kayak développe un programme Python pour gérer les résultats lors d'une compétition.
Dans ce programme, pour modéliser les concurrents et leurs résultats, une classe `Concurrent` est définie avec les attributs suivants :

- `nom` de type `Str` qui représente le pseudonyme du compétiteur ;
- `temps` de type `Float` qui est le temps mis pour réaliser le parcours en secondes ;
- `penalites` de type `Int` qui est le nombre de secondes de pénalité cumulées octroyées au concurrent ;
- `temps_tot` de type `Float` qui correspond au temps total, c'est-à-dire au temps mis pour réaliser le parcours auquel on a ajouté les pénalités.

On suppose que tous les concurrents ont des temps différents dans cet exercice.

Le code Python incomplet de la classe `Concurrent` est donné ci-dessous.

```
class Concurrent:
    def __init__(self, pseudo, temps, penalite):
        self.nom = pseudo
        self.temps = temps
        self.penalite = ...
        self.temps_tot = ...
```

1. (a) Recopier et compléter le code du constructeur de la classe `Concurrent`.

   On exécute l'instruction suivante : `c1 = Concurrent("Mosquito", 87.67, 12)`

   (b) Donner la valeur de l'attribut `temps_tot` de `c1`.

   (c) Donner l'instruction permettant d'accéder à la valeur `temps_tot` de `c1`.

2. Pendant la course, des instances de la classe `Concurrent` sont créées au fur et à mesure des arrivées des concurrents.
   On définit une classe `Liste` pour les stocker au fur et à mesure. Cette classe implémente la structure de données abstraite liste dont l'interface est munie :

   — du constructeur qui ne prend pas de paramètre et qui crée une liste vide.
   Exemple : `L = Liste()`

   — de la méthode `est_vide` qui ne prend pas de paramètre et qui renvoie un booléen : `True` si la liste est vide ou `False` sinon.
   Exemple : On considère la liste `L = <c1, c2, c3>` ou `c1`, `c2` et `c3` sont des instances de `Concurrent`.
   L'appel `L.est_vide()` renvoie `False`.

— de la méthode `tete` qui ne prend pas de paramètre et qui renvoie un objet de type `Concurrent` ayant pour valeur le premier élément de la liste. Cet élément sera appelé tête de la liste dans la suite de l'exercice.

Cette méthode ne s'applique que sur des listes non vides.

Exemple : On considère la liste `L = <c1, c2, c3>` ou `c1`, `c2` et `c3` sont des instances de `Concurrent`.

`L.tete()` a pour valeur `c1`.

Remarque : Après exécution de `L.tete()`, la liste `L` reste inchangée et vaut toujours `<c1, c2, c3>`.

— de la méthode `queue` qui ne prend pas de paramètre. Cette méthode renvoie la liste sur laquelle elle s'applique privée de son premier élément.

Cette méthode ne s'applique que sur des listes non vides.

Exemple : On considère la liste `L = <c1, c2, c3>`

L'appel `L.queue()` renvoie la liste `<c2, c3>`

Remarque : Après exécution de `L.queue()`, la liste `L` reste inchangée et vaut toujours `<c1, c2, c3>`.

— de la méthode `ajout` qui prend en paramètre un concurrent `c` et qui modifie la liste sur laquelle elle s'applique, en ajoutant `c` en tête.

Exemple 1 :

Si `L` est la liste vide, `L.ajout(c)` modifie la liste `L` qui devient `<c>`

Exemple 2 :

Si `L` est la liste `<c1, c2, c3>`, `L.ajout(c)` modifie la liste `L` qui devient `<c, c1, c2, c3>`

On considère le script Python suivant :

```
1 c1= Concurrent("Mosquito",87.67,12)
2 c2= Concurrent("Python Fute",89.73,4)
3 c3= Concurrent("Piranha Vorace",90.54,0)
4 c4= Concurrent("Truite Agile",84.32,52)
5 c5= Concurrent("Tortue Rapide",92.12,2)
6 c6= Concurrent("Lievre Tranquille",93.45,0)
7
8 resultats=Liste()
9 resultats.ajout(c1)
10 resultats.ajout(c2)
11 resultats.ajout(c3)
12 resultats.ajout(c4)
13 resultats.ajout(c5)
14 resultats.ajout(c6)
```

Après exécution, ce script génére une liste `resultats` que l'on peut représenter par :

`<c6, c5, c4, c3, c2, c1>`

(a) On considère la liste `resultats` ci-dessus.
Donner la ou les instruction(s) qui permet(tent) d'accéder à `c4`.

(b) Donner la ou les instruction(s) qui permet(tent) d'accéder au temps total du concurrent stocké en tête de la liste `resultats`.

3. On souhaite créer une fonction `meilleur_concurrent` qui prend en paramètre une liste `L` de concurrents et qui renvoie l'objet `Concurrent` correspondant au concurrent le plus rapide. On suppose que la liste est non vide.

Recopier et compléter le code Python, donné ci-dessous, de la fonction `meilleur_concurrent`.

```
def meilleur_concurrent(L) :
        conc_mini = L.  ...
        mini = conc_mini.temps_tot
        Q = L.queue()
        while not(Q.est_vide()):
                elt = Q.tete()
                if elt.temps_tot ... mini :
                        conc_mini = elt
                        mini = elt.temps_tot
                Q = Q.  ...
        return ...
```

4. Pour simplifier le stockage des résultats, on décide de stocker les objets de la classe `Concurrent` dans un arbre binaire de recherche. Chaque nœud de cet arbre est donc un objet `Concurrent`.

Dans cet arbre binaire de recherche, en tout nœud :

— le concurrent enfant à gauche est plus rapide que le nœud ;

— le concurrent enfant à droite est moins rapide que le nœud.

Pour implémenter la structure d'arbre binaire de recherche, on dispose d'une classe `Arbre` munie, entre autres, d'une méthode `ajout` qui prend en paramètre un objet `c` de type `Concurrent` et qui modifie l'arbre binaire sur lequel elle s'applique, en y ajoutant le concurrent `c` tout en maintenant la propriété d'arbre binaire de recherche.

On ajoute dans un arbre vide successivement les concurrents de la liste `resultats` en partant de la tête de la liste (soit, dans le cas présent, `c6`, puis `c5`, puis `c4`,...).

Dessiner l'arbre binaire de recherche obtenu. On rappelle le temps total de chaque concurrent :

Concurrent	c6	c5	c4	c3	c2	c1
temps_tot	93,45	94,12	136,32	90,54	93,73	99,67

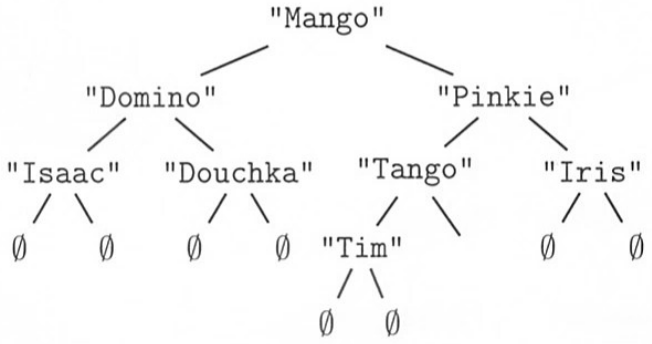
Exercice 4 (4 points).

Cet exercice porte sur les arbres binaires et leurs algorithmes associés.

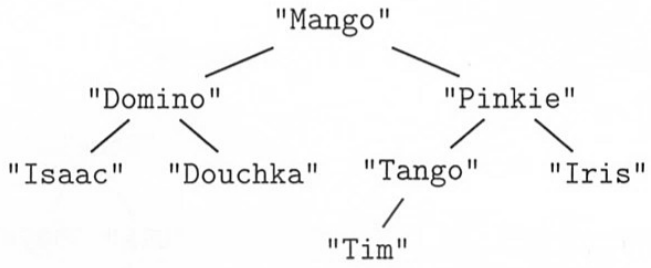
Un éleveur de chiens gère les informations sur ses animaux à l'aide d'un logiciel qui mémorise le pédigrée de chacun de ses chiens. Le pédigrée d'un chien correspond à son arbre généalogique.
Une structure **arbre de pédigrée** est définie récursivement, soit par un arbre vide, noté $\emptyset$, soit par un arbre binaire où :

- la valeur du nœud est une chaîne de caractères qui représente le nom de l'animal ;
- le sous-arbre gauche est l'arbre de pédigrée du père du chien ;
- le sous-arbre droit est l'arbre de pédigrée de la mère du chien.

On représente donc graphiquement un arbre de pédigrée comme l'arbre `A` suivant :



Pour alléger la représentation d'un arbre de pédigrée, on ne notera pas les arbres vides, l'arbre précédent sera donc représenté comme ci-dessous.

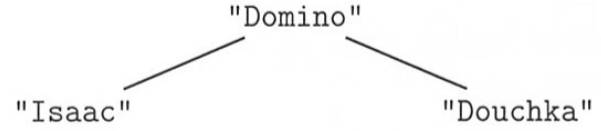


Dans cet arbre,

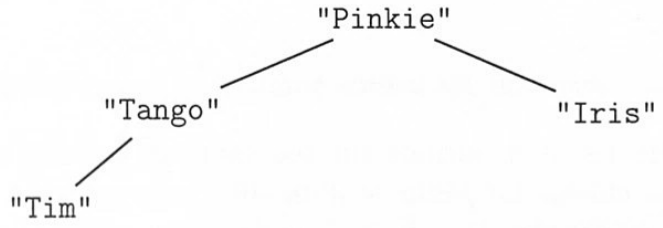
- le père de Mango est Domino et sa mère Pinkie ;
- les parents de Douchka ne sont pas connus ;
- Iris est la mère de Pinkie ;
- la mère de Tango n'est pas connue.

Pour manipuler les arbres de pédigrée, on dispose des quatre fonctions suivantes :

- La fonction `racine` qui prend en paramètre un arbre de pédigrée non vide et renvoie la valeur de la racine.
  Exemple : en reprenant l'exemple d'arbre de pédigrée `A`, `racine(A)` vaut `"Mango"`.
- La fonction `gauche` qui prend en paramètre un arbre de pédigrée non vide et renvoie son sous-arbre gauche correspondant à l'arbre de pédigrée du père.
  Exemple : en reprenant l'exemple d'arbre de pédigrée `A`, `gauche(A)` est l'arbre représenté graphiquement ci-après :



- La fonction `droit` qui prend en paramètre un arbre de pédigrée non vide et renvoie son sous-arbre droit correspondant à l'arbre de pédigrée de la mère.
  Exemple : en reprenant l'exemple d'arbre de pédigrée `A`, `droit(A)` vaut l'arbre représenté graphiquement ci-après :

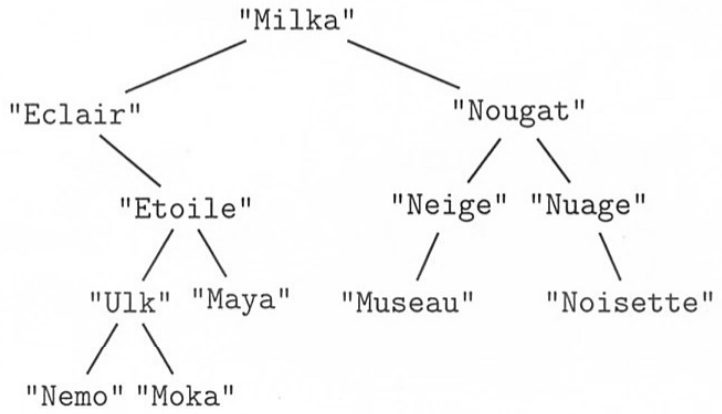


— La fonction `est_vide` qui prend en paramètre un arbre de pédigrée et renvoie `True` si l'arbre est vide ou `False` sinon.

Exemple : en reprenant l'exemple d'arbre de pédigrée `A`, `est_vide(A)` vaut `False`.

Pour toutes les questions de l'exercice, on suppose que tous les chiens d'un même pédigrée ont un nom différent.

1. On considère l'arbre de pédigrée `B` suivant :



(a) Déterminer la valeur de la racine de cet arbre.

(b) On appelle feuille d'un arbre de pédigrée, un nœud dont les sous-arbres gauche et droit sont vides.

Déterminer l'ensemble des valeurs des feuilles de cet arbre.

(c) Déterminer si `"Nuage"` est un mâle ou une femelle.

(d) Déterminer le père et la mère de `"Etoile"`.

2. (a) Recopier et compléter la fonction récursive Python `present` ayant pour paramètres un arbre de pédigrée `arb` et le nom d'un chien `nom` et qui renvoie `True` si ce nom est présent dans l'arbre de pédigrée ou `False` sinon.

```
def present(arb, nom):
    if est_vide(arb):
        return ...
    elif racine(arb) == ... :
        return ...
    else:
        return present( ... , ... ) or present( ... , ... )
```

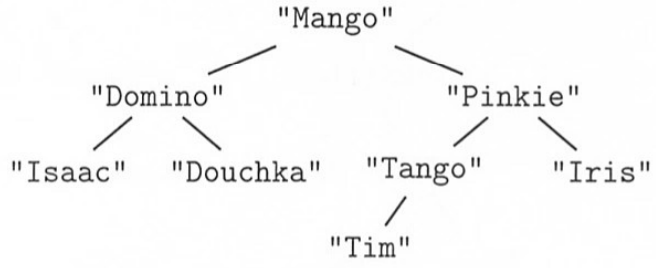
Pour toute la suite de l'exercice, on pourra utiliser la fonction `present` même si la question 2.(a) n'a pas été traitée.

(b) Écrire une fonction Python `parents` ayant pour paramètre un arbre de pédigrée `arbre` d'un chien et qui renvoie le p-uplet `parents` des deux parents de ce chien dans l'ordre père, mère. Si un des parents est inconnu, il sera noté `""`.

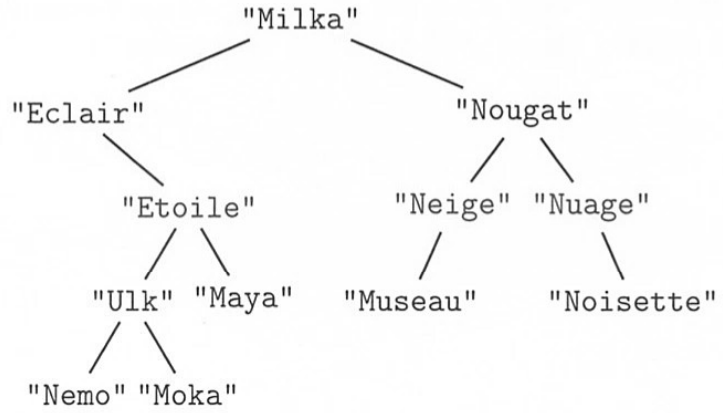
Exemple : `parents(B)` vaut `("Eclair","Nougat")`

3. (a) On dit que deux chiens sont frères et sœurs s'ils ont le même père ou la même mère.
On considère les trois arbres de pédigrée suivants :

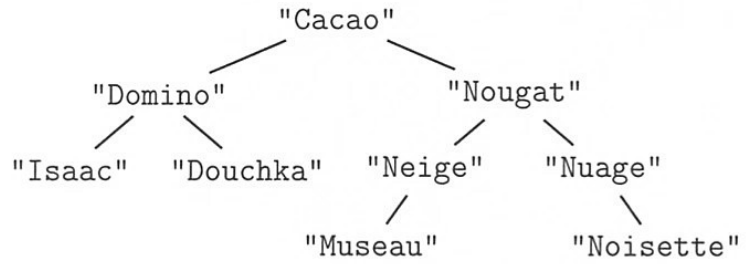
Arbre A :



Arbre B :



Arbre C :



Parmi les trois chiens Mango, Milka et Cacao, déterminer les liens de fratrie.

(b) Écrire une fonction Python `frere_soeur` ayant pour paramètres deux arbres de pédigrée `arbre1` et `arbre2` correspondant à deux chiens. Cette fonction renvoie `True` si les deux chiens ont le même père ou la même mère ou `False` sinon.

4. Étant donné l'arbre de pédigrée d'un chien, on considère que :

— le niveau 0 est le niveau de la racine contenant le nom du chien ;

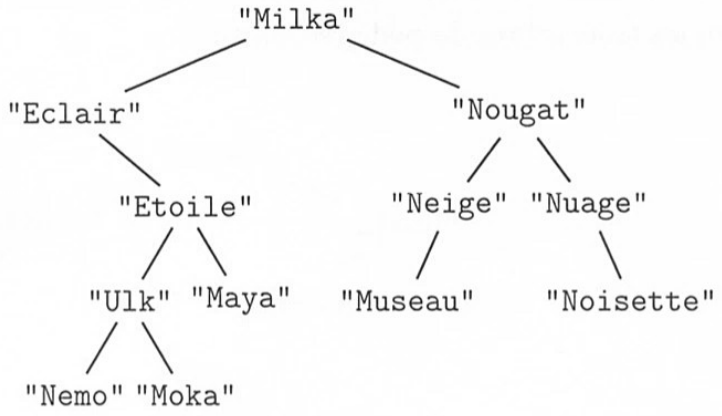
— le niveau 1 est le niveau des parents du chien ;

— le niveau 2 est le niveau des grands-parents du chien ;

— etc.

Proposer une fonction Python `nombre_chiens` ayant pour paramètres un arbre de pédigrée `arb` et un entier `n` et qui renvoie le nombre de noms connus dans l'arbre de pédigrée `arb` au niveau `n`.

Exemple : On considère l'arbre de pédigrée `B` suivant :



`nombre_chiens(B,3)` vaut 4 car les noms des chiens mentionnés dans l'arbre de pédigrée au niveau 3 sont `"Ulk"`, `"Maya"`, `"Museau"` et `"Noisette"`.