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.

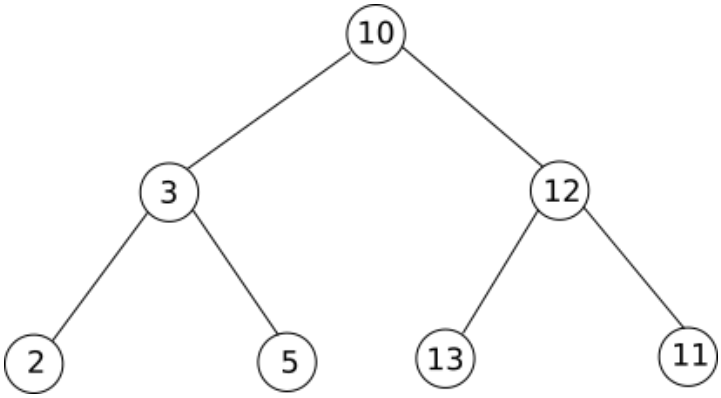


Figure 1

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

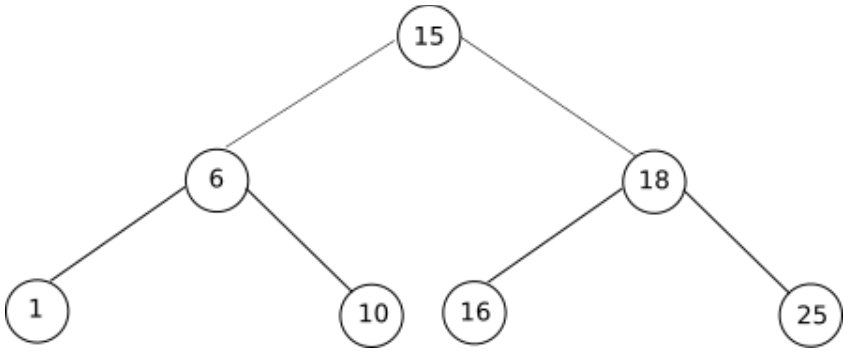


Figure 2

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.

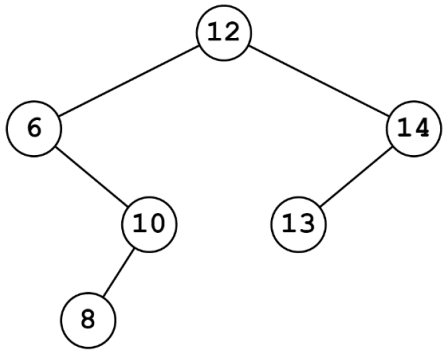


Figure 1 : Exemple d'un arbre binaire

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 :

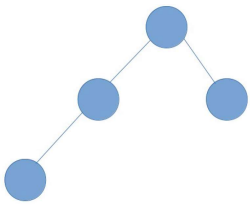
- `est_vide(A)` qui renvoie `True` si l'arbre binaire `A` est vide, `False` s'il ne l'est pas ;
- `sous_arbre_gauche(A)` qui renvoie le sous-arbre gauche de l'arbre binaire `A` ;
- `sous_arbre_droit(A)` qui renvoie le sous-arbre droit de l'arbre binaire `A`.

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.

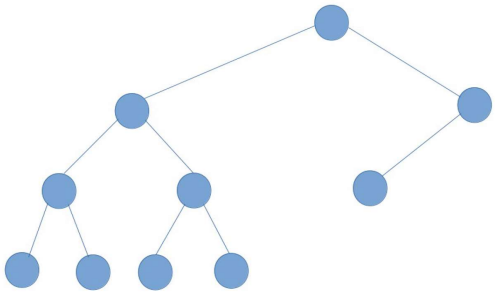
   Indication : $2^{h+1}-1 = 1+2+4+...+2^h$.

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.