IV - TP 4 Arbres binaires et arbres binaire de recherche

1 Arbre binaire : implémentation

1.1 Modélisation

On rappelle qu'un arbre binaire est constitué d'un ensemble (éventuellement vide) de nœuds. Si l'arbre binaire n'est pas vide, il possède un nœud particulier nommé **la racine** de l'arbre, munie la plupart du temps d'une **étiquette**. Les nœuds restants sont alors répartis entre le **sous-arbre gauche** et **le sous-arbre droit**. Nous utiliserons dans ce TP deux classes pour implémenter les objets de type `Arbre`. La classe `Noeud` permet de représenter les nœuds de l'arbre. Il est à noter que l'étiquette sera toujours un entier dans ce TP, et que les attributs `gauche` et `droit` d'un nœud seront des objets de type `ArbreBinaire` dans cette partie.

Code

```
class Noeud:
    def __init__(self, v, g, d):
        self.etiquette = v
        self.gauche = g
        self.droit = d
```

La seconde classe `ArbreBinaire` permet de représenter un arbre binaire. Un objet de type `ArbreBinaire` possède un unique attribut, `racine`, de type `Nœud`. On prendra pour convention que lorsque l'attribut `racine` vaut la valeur spéciale `None`, cela correspond à l'arbre vide. On remarquera l'utilisation de l'argument optionnel `n` dans la définition de la fonction `__init__`. Ainsi, il est possible d'instancier une variable `a` de type `ArbreBinaire` de deux manières différentes :

- `a = ArbreBinaire()` : `a.racine` vaut alors `None`
- `a = ArbreBinaire(n)`, avec `n` un objet de type `Noeud`. `a.racine` est alors `n`.

Code

```
class ArbreBinaire:
    """Représente un arbre binaire"""
    def __init__(self, n = None):
        """ArbreBinaire, Noeud -> NoneType"""
        self.racine = n

```

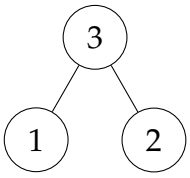
On donne ci-dessous des exemples d'utilisation des classes `Noeud` et `ArbreBinaire`.

Code

```
a_vide = ArbreBinaire()
feuille1 = ArbreBinaire(Noeud(1,
                              ArbreBinaire(),
                              ArbreBinaire()))
feuille2 = ArbreBinaire(Noeud(2,
                              ArbreBinaire(),
                              ArbreBinaire()))
a0 = ArbreBinaire(Noeud(3, feuille1, feuille2))
```

Avec le code donné dans le sujet, il est à noter que :

- `ArbreBinaire()` renvoie un objet de type `ArbreBinaire` dont l'attribut `racine` vaut `None` : il s'agit d'un arbre binaire vide.
- La variable `feuille1` est un objet de type `ArbreBinaire`. Sa racine est un objet de type `Noeud`, dont **les attributs `gauche` et `droit` sont des objets de type `ArbreBinaire`**.
- Enfin, la variable `a0` correspond à l'arbre binaire ci-dessous :



Arbre $\mathcal{A}_0$

1.2 Méthodes de l'interface

Compléter le code python des fonctions de l'interface du type `ArbreBinaire`, telles que décrites dans le cours.
Remarque. Si $\mathcal{A}$ est un arbre binaire de racine `r`, et `s1` (resp. `s2`) la chaîne de caractère représentant le sous-arbre gauche (resp. droit) de $\mathcal{A}$, alors on représentera $\mathcal{A}$ à l'aide de la chaîne de caractères :

```
({s1}) {r.etiquette} ({s2})
```

L'arbre vide sera représenté par la chaîne de caractère vide.

Code

```python
def est_vide(self):
    """ ArbreBinaire -> bool """
    return ...

def etiquette(self):
    """ ArbreBinaire -> int """
    return ...

def gauche(self):
    """ ArbreBinaire -> ArbreBinaire """
    return ...

def droit(self):
    """ ArbreBinaire -> ArbreBinaire """
    return ...

def est_feuille(self):
    """ ArbreBinaire -> bool """
    return ...

def __str__(self):
    """ ArbreBinaire -> str """
    pass
```

Code

```python
print(a0)
```

Résultat

```
(() 1 ()) 3 (() 2 ())
```

1.3 Méthode `ajouter_depuis_tableau`

On souhaite dans cette question écrire une méthode efficace pour créer un arbre binaire dont on fournit la liste la liste des étiquettes. `tab` est un tableau dont les éléments sont des entiers ou bien la valeur spéciale `None`, tel qu'obtenu avec un parcours en largeur de l'arbre binaire souhaité.

À l'aide du tableau `tab`, l'algorithme (**similaire à celui d'un parcours en largeur**) pour reconstruire l'arbre binaire `self` correspondant est le suivant :

- l'arbre binaire initialement présent dans la file est `self`.
- tant que `tab` n'est pas vide :
  - soit `a` le premier arbre de la file ;
  - soit `elem` le premier élément du tableau `tab`.
    - Si celui-ci n'est pas `None` : on affecte à la racine de `a` un `Noeud` d'étiquette `elem` et dont les deux enfants sont des arbres binaires vides.
    - Sinon on ne fait rien.
  - `elem` a été ajouté à l'arbre binaire en construction. On supprime donc `elem` de `tab`.
  - on ajoute les enfants gauche et droit de l'arbre `a` à la file dans le cas où `a` n'est pas l'arbre vide.

Code

```
def ajouter_depuis_tableau(self, tab):
    """ ArbreBinaire, [int] -> Nonetype
    Ajoute à l'arbre binaire self (supposé initiallement vide) les
    éléments de tab (correspondant à un parcours en largeur de l'arbre)
    """
    pass
```

Code

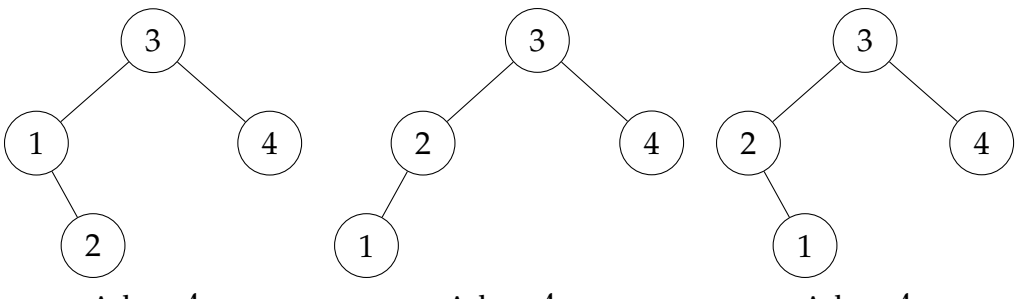
```
a1 = ArbreBinaire()
tab = [1, 2, 3, 4, None, None, 6]
a1.ajouter_depuis_tableau(tab)
print(a1)
```

Résultat

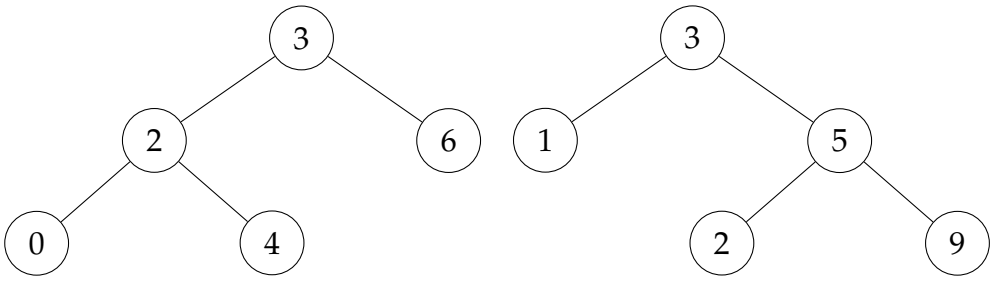
```
((() 4 ()) 2 ()) 1 (() 3 (() 6 ()))
```

Question 1.

1. Quel est "l'opération de traitement" effectuée chaque nœud de l'arbre ?
2. Dessiner l'arbre binaire correspondant à la variable `a1` de l'exemple précédent.
3. Utiliser la méthode `ajouter_depuis_tableau` pour instancier des variables `a2`, ..., `a6` correspondant aux arbres suivants :



Arbre $\mathcal{A}_2$ Arbre $\mathcal{A}_3$ Arbre $\mathcal{A}_4$



Arbre $\mathcal{A}_5$ Arbre $\mathcal{A}_6$

1.4 Méthode `taille`

Écrire une méthode `taille` de la classe `ArbreBinaire` qui renvoie le nombre de nœuds dont est constitué l'objet `self`.

Code

```
def taille(self):
    """ ArbreBinaire -> int
    Renvoie le nombre de nœuds dont est composé self """
    pass
```

Code

```
for a in [a0, a1, a2, a3, a4, a5, a6]:
    print(a.taille(), end = " ")
```

Résultat

```
3 5 4 4 4 5 5
```

1.5 Méthode `appartient`

Écrire une méthode `appartient` de la classe `ArbreBinaire` qui détermine si l'élément `e` est une étiquette d'un des nœuds de `self`.

Code

```
def appartient(self, e):
    """ ArbreBinaire, int -> bool
    Détermine si l'entier e est une étiquette de l'arbre self """
    pass
```

Code

```
for a in [a0, a1, a2, a3, a4, a5, a6]:
    print(a.appartient(4), end = " ")
```

Résultat

```
False True True True True True False
```

1.6 Méthode `inserer`

Écrire une méthode `inserer` qui insère l'élément `e` dans l'arbre `self` de la manière suivante :

- si `self` est vide, alors on affecte à son attribut `racine` un nœud d'étiquette `e` ;
- sinon, on insère récursivement `e` dans le sous-arbre gauche ou droit de `self` qui possède le moins d'éléments (on pourra utiliser la méthode `taille`).

Code

```
def inserer(self, e):
    """ ArbreBinaire, int
    Insère l'entier e dans l'arbre binaire self """
    pass
```

Code

```
a7 = ArbreBinaire()
for i in range(7):
    a7.inserer(i)
print(a7)
```

Résultat

```
((() 3 ()) 1 (() 5 ())) 0 ((() 4 ()) 2 (() 6 ()))
```

Question 2. **1.** Quelle est la hauteur de l'arbre `a7` ?

2. Combien peut-on encore insérer d'élément avec la méthode `inserer` avant que la hauteur de `a7` ne soit strictement supérieure à 3 ?

3. Quelle est la propriété que possède `a` de type `ArbreBinaire` si tous les éléments qui le composent y ont été ajoutés à l'aide de la méthode `inserer` ?

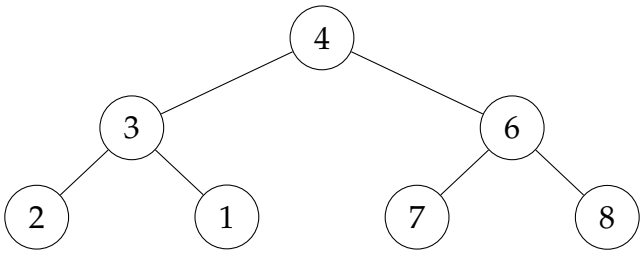
2 Arbre binaire de recherche

2.1 Définition

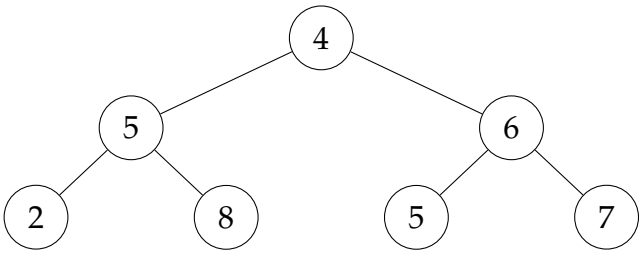
Définition 1. On appelle arbre binaire de recherche (ABR) un arbre binaire étiqueté qui vérifie les propriétés suivantes :

- toutes les étiquettes de l'arbre appartiennent au même ensemble, et peuvent être comparées entres elles ;
- le sous-arbre gauche est un arbre binaire de recherche dont toutes les étiquettes sont inférieures ou égales à l'étiquette de la racine ;
- le sous-arbre droit est un arbre binaire de recherche dont toutes les étiquettes sont supérieures ou égales à l'étiquette de la racine.

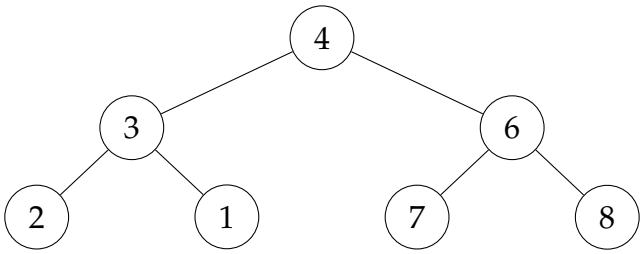
Question 3. **1.** Parmi les arbres suivants, lesquels sont des arbres binaires de recherche ?



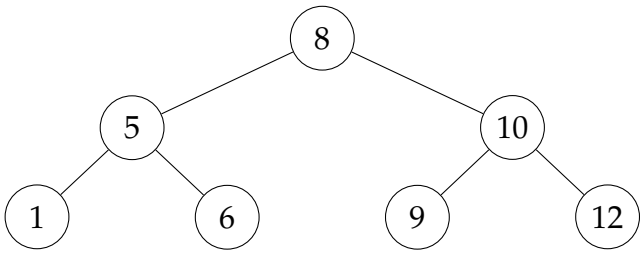
Arbre $\mathcal{A}_7$



Arbre $\mathcal{A}_8$



Arbre $\mathcal{A}_9$



Arbre $\mathcal{A}_{10}$

2. Citer les arbres binaires de recherche parmi les arbres $\mathcal{A}_1, \ldots, \mathcal{A}_6$.

2.2 Implémentation

Un ABR est un arbre, dont les nœuds sont répartis de manière particulière. Ainsi, toutes les propriétés, toutes les méthodes s'appliquant aux arbres binaires s'appliquent également aux ABR. Attention cependant, la réciproque est fausse ! Si tout ABR est un arbre binaire, tout arbre binaire n'est pas un ABR.

Nous verrons que la structure d'ABR permet notamment d'optimiser certaines opérations, notamment la méthode `appartient`. On souhaite écrire une classe `ABR` afin de manipuler les arbres binaires de recherche. On pourrait copier/coller la classe `ArbreBinaire`, mais on aurait alors dupliqué notre code (c'est toujours une mauvaise idée). On utilise plutôt le concept d'**héritage**.

Code

```
class ABR(ArbreBinaire):
    """ Représentation d'ABR (Arbre Binaire de Recherche) """
    pass
```

On dit avec cette syntaxe que la classe `ABR` **hérite** de la classe `ArbreBinaire` : les objets de type `ABR` possèdent toutes les méthodes des objets de type `ArbreBinaire` (en particulier la méthode `__init__` : leurs attributs sont donc également les mêmes).

2.2.1 Exemple

On donne ci-dessous un exemple d'utilisation de la classe `ABR`.

Code

```
abr_vide = ABR()
feuille1 = ABR(Noeud(1,
                     ABR(),
                     ABR()))
feuille2 = ABR(Noeud(3,
                     ABR(),
                     ABR()))
a11 = ABR(Noeud(2, feuille1, feuille2))

print(a11)
print(a11.taille())
print(a11.appartient(4))
```

Résultat

```
(() 1 ()) 2 (() 3 ())
3
False
```

2.3 Méthode `inserer`

Écrire la méthode `inserer` de la classe `ABR` qui insère l'élément `e` **au bon endroit** dans l'ABR `self` : en particulier, `self` sera toujours un ABR à l'issue de l'exécution de la méthode `inserer`.

Code

```
def inserer(self, e):
    """ ABR, int -> NoneType
    Insère l'élément e dans l'ABR self """
    pass
```

Code

```
a11.inserer(5)
a11.inserer(0)
print(a11)
```

Résultat

```
((() 0 ()) 1 ()) 2 (() 3 (() 5 ()))
```

2.4 Méthode `appartient`

Écrire une méthode `appartient` de la classe `ABR` qui détermine si l'élément `e` est une étiquettes d'un des nœuds de `self`. On fera attention à ne **réaliser au plus qu'un seul appel récursif.**

Code

```
def appartient(self, e):
    """ ABR, int -> Nonetype
    Détermine si l'élément e est une des étiquettes de l'ABR self """
    pass
```

Code

```
print(a11.appartient(5))
print(a11.appartient(0))
print(a11.appartient(50))
```

Résultat

```
True
True
False
```

2.5 Méthode `minimum`

Écrire une méthode `minimum` de la classe `ABR` qui renvoie la plus petite étiquette présent dans `self`. On fera attention à ne **réaliser au plus qu'un seul appel récursif.**

Code

```
def minimum(self):
    """ ABR -> int
    Renvoie la plus petite étiquette de self """
    pass
```

Code

```
print(a11.minimum())
```

Résultat

```
0
```

2.6 À propos de l'héritage

Question 4. **1.** Commenter l'affichage suivant :

Code

```
print(a11.taille)
print(a11.ajouter_depuis_tableau)
print(a11.appartient)
print(a11.minimum)
```

Résultat

```
<bound method ArbreBinaire.taille of <__main__.ABR object at
 ↪  0x7fa720d86950>>
<bound method ArbreBinaire.ajouter_depuis_tableau of <__main__.ABR
 ↪  object at 0x7fa720d86950>>
<bound method ABR.appartient of <__main__.ABR object at
 ↪  0x7fa720d86950>>
<bound method ABR.minimum of <__main__.ABR object at
 ↪  0x7fa720d86950>>
```

2. Commenter l'affichage :

Code

```
a12 = ABR()
a12.ajouter_depuis_tableau([2, 1, 3])
print(a12)
a12.inserer(5)
print(a12)
```

Résultat

```
(() 1 ()) 2 (() 3 ())
(() 1 ()) 2 ((() 5 ()) 3 ())
```

3. Proposer une méthode `ajouter_depuis_tableau` de la classe `ABR` qui étant donné un ABR `self` supposé vide le rempli avec les éléments de `tab`. À l'issue de l'exécution de cette méthode, `self` est toujours un ABR.