Implémentation des arbres en Python

S4 - Arbres et graphes

L'objectif de cette partie est d'**implémenter** la structure d'arbre binaire en Python. Nous allons pour cela utiliser la Programmation Orientée Objet et construire un module réutilisable proposant à l'utilisateur une interface (**API**) permettant de travailler avec les arbres binaires.

1. Arbres binaires

Une interface souhaitable devrait permettre de :

- Créer un arbre vide ;
- Accéder au sous-arbre gauche et au sous-arbre droit d'un nœud ;
- Accéder à une clef ;
- Tester si un nœud est une feuille ;
- Tester si un arbre est vide ;
- Retourner la taille ;
- Retourner la hauteur.

De plus, il serait souhaitable de parvenir à afficher un arbre de façon visuelle.

Nous avons vu que la structure d'arbre binaire est une structure **récursive** : cette propriété est exploitée dans l'implémentation que nous allons présenter. Pour définir un arbre, il suffit de définir un nœud racine ainsi que les deux sous-arbres gauche et droite qui sont eux-même des arbres binaires. Cela revient à assimiler un arbre à sa racine associée à un lien vers ses deux fils.

Nous définissons ci-dessous un objet `ArbreBinaire` possédant trois attributs `clef`, `gauche`, `droit`. Pour respecter les principes de la POO, et notamment la notion d'**encapsulation**, nous avons défini des méthodes d'accès aux attributs (elles commencent par `get`) et des méthodes de modification des attributs (elles commencent par `set`) et on s'interdira tout accès ou affectation direct(e) du type `arbre.racine = ...`.

La méthode `setRacine`, qui permet de définir la clef d'un nœud assure que chaque nœud a toujours un sous-arbre gauche **et** un sous-arbre droit, éventuellement vides, ce qui facilite le traitement des arbres dans les algorithmes suivants. On matérialise ici l'aspect récursif de la structure.

```
class ArbreBinaire:
    """ Implémentation de la structure d'arbre binaire """

    def __init__(self):
        self.racine = None
        # les sous-arbres gauche et droit doivent être des
        # instances de l'objet ArbreBinaire
        self.gauche = None
        self.droit = None
```

```
    def setRacine(self, racine):
        """définit la clef de la racine de l'instance
         et crée les sous arbres vides gauches et droits"""
        self.racine = racine
        if self.gauche is None:
            self.gauche = ArbreBinaire()
        if self.droit is None:
            self.droit = ArbreBinaire()

    def getRacine(self):
        """retourne la clef de la racine de l'arbre"""
        return self.racine

    def getSousArbreGauche(self):
        return self.gauche

    def setSousArbreGauche(self, arbre):
        if isinstance(arbre, ArbreBinaire):
            self.gauche = arbre

    def getSousArbreDroit(self):
        return self.droit

    def setSousArbreDroit(self, arbre):
        if isinstance(arbre, ArbreBinaire):
            self.droit = arbre

    def estVide(self) -> bool:
        return self.racine is None

    def estFeuille(self) -> bool:
        if self.estVide():
            return False
        else:
            return self.gauche.estVide() and self.droit.estVide()

    def __str__(self):
        if self.estVide():
            return "()"
        elif self.estFeuille():
            return f"('{self.racine}', (), ())"
        else:
            return f"('{self.racine}', {self.gauche.__str__()}, {self.droit.__str__()})"
```

La classe est complétée par une méthode `estVide` permettant de tester si un arbre est vide ou non et une méthode `estFeuille` permettant de tester si un nœud est une feuille ou non (on confond un nœud avec un arbre de hauteur 1).

La dernière méthode est la méthode spéciale `__str__` qui définit la façon dont un arbre va être affiché par la fonction `print`. Ici, on a choisi un affichage sous forme de tuple du type `(clef, sous-arbre gauche,`

`sous-arbre droit`).

Pour créer un module, on enregistre le code ci-dessus dans un fichier nommé par exemple `structures.py`.

On peut ensuite utiliser notre nouvelle structure dans un autre fichier Python (dans le même dossier), ou dans la console interactive, en important le module :

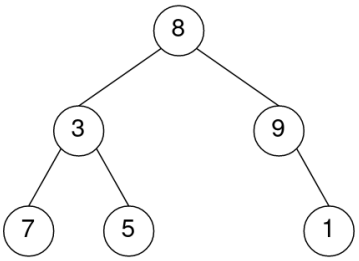
```
from structures import *

a = ArbreBinaire()
a.setRacine(8)
a.getSousArbreGauche().setRacine(3)
a.getSousArbreDroit().setRacine(9)
b = a.getSousArbreGauche()
c = a.getSousArbreDroit()
b.getSousArbreGauche().setRacine(7)
b.getSousArbreDroit().setRacine(5)
c.getSousArbreDroit().setRacine(1)
print(a)
```

On obtient en sortie :

```
>>> (8, (3, (7, (), ()), (5, (), ())), (9, (), (1, (), ())))
```

Cela correspond à l'arbre représenté ci-dessous :



On peut tester les autres méthodes dans la console :

```
print(c)
>>> (9, (), (1, (), ()))
c.getSousArbreGauche().estVide()
>>> True
c.estFeuille()
>>> False
c.getSousArbreDroit().estFeuille()
>>> True
```

Nous pouvons maintenant ajouter au fichier `structures.py` les deux fonctions suivantes (en dehors de la classe `ArbreBinaire` car ce ne sont pas des méthodes) qui retournent respectivement la taille et la hauteur d'un arbre binaire.

```
def taille(arbre) -> int:
    """Retourne la taille de l'arbre, càd son nombre de noeuds"""
    if arbre.racine is None:
        return 0
    else:
        return 1 + taille(arbre.gauche) + taille(arbre.droit)

def hauteur(arbre) -> int:
    """Retourne la hauteur de l'arbre"""
    if arbre.racine is None:
        return 0
    else:
        return 1 + max(hauteur(arbre.gauche), hauteur(arbre.droit))
```

Prendre le temps de bien comprendre comment fonctionnent ces deux fonctions ...

```
taille(a)
>>> 6
hauteur(a)
>>> 3
```

Ce module `structures` sera utilisé en exercices et plus tard dans l'année lorsque nous étudierons les algorithmes sur les arbres.

2. Arbres binaires de recherche (ABR)

Les ABR sont des arbres binaires. Nous pouvons donc créer une classe `ABR` fille de la classe `ArbreBinaire` en utilisant la notion d'**héritage** et de **polymorphisme** de la POO (voir les compléments de cours à ce sujet). Nous définissons une méthode spécifique : l'insertion d'une clef. Cette méthode ajoute une clef à un ABR existant en s'assurant que l'arbre obtenu est toujours un ABR (le nouveau nœud est toujours une feuille).

```
class ABR(ArbreBinaire):
    """ Implémentation de la structure d'arbre binaire de recherche """

    def __init__(self):
        super().__init__()

    def setRacine(self, racine):
        """définit la clef de la racine de l'instance
         et crée les sous arbres vides gauches et droits
         Provoque une erreur si la racine casse la structure d'ABR"""
        self.racine = racine
        if self.gauche is None:
            self.gauche = ABR()
        if self.droit is None:
            self.droit = ABR()
        if not estABR(self):
```

```
            raise Exception("Cette affectation de clef casse la structure ABR !!!")

    def insere(self, racine):
        """insère une clef dans l'arbre en préservant la structure ABR"""
        if self.racine is None:
            self.racine = racine
            self.gauche = ABR()
            self.droit = ABR()
        else:
            if racine < self.racine:
                self.gauche.insere(racine)
            else:
                self.droit.insere(racine)
```

Pour définir un arbre binaire de recherche valide, on utilisera toujours la méthode `insere` car elle permet de s'assurer de toujours conserver un ABR.

Pour faciliter la vérification, nous définissons une fonction `estABR` qui peut s'appliquer aussi bien à un arbre binaire quelconque qu'à un ABR et qui retourne `True` si l'arbre est un ABR et `False` sinon.

```
def estABR(arbre, mini=-float("inf"), maxi=+float("inf")) -> bool:
    if arbre.getRacine() is None:
        return True
    else:
        return estABR(arbre.getSousArbreGauche(), mini, arbre.getRacine()) and
               estABR(arbre.getSousArbreDroit(), arbre.getRacine(), maxi) and
               mini < arbre.racine < maxi
```

Prendre le temps de bien comprendre cette fonction …

Utilisation :

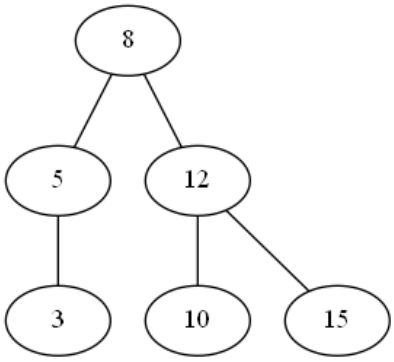
```
from structures import *

a = ABR()
a.setRacine(8)
a.insere(5)
a.insere(3)
a.insere(12)
a.insere(10)
a.insere(15)
print(a)
print(estABR(a))
# Affectation directe à proscrire :
# a.getSousArbreDroit().setRacine(1) ## provoque une erreur
```

Sortie :

```
(8, (5, (3, (), ()), ()), (12, (10, (), ()), (15, (), ())))
True
```

L'arbre correspond à :



Le module `structure.py` est à conserver : il sera utilisé en exercices et dans les chapitres suivants.

Complément

On peut ajouter une fonctionnalité de représentation graphique d'un arbre en utilisant les bibliothèques `networkx` et `matplotlib`. Ajouter la fonction ci-dessous au fichier `structures.py` :

```
import networkx as nx
import matplotlib.pyplot as plt

def afficheArbre(arbre, size=(4,4), null_node=False):
"""
size : tuple de 2 entiers. Si size est int -> (size, size)
null_node : si True, trace les liaisons vers les sous-arbres vides
"""
arbreAsTuple = eval(arbre.__str__())
def parkour(arbre, noeuds, branches, labels, positions, profondeur,
            pos_courante, pos_parent, null_node):
    if arbre != ():
        noeuds[0].append(pos_courante)
        positions[pos_courante] = (pos_courante, profondeur)
        profondeur -= 1
        labels[pos_courante] = str(arbre[0])
        branches[0].append((pos_courante, pos_parent))
        pos_gauche = pos_courante - 2 ** profondeur
        parkour(arbre[1], noeuds, branches, labels, positions, profondeur,
                pos_gauche, pos_courante, null_node)
        pos_droit = pos_courante + 2 ** profondeur
        parkour(arbre[2], noeuds, branches, labels, positions, profondeur,
                pos_droit, pos_courante, null_node)
    elif null_node:
        noeuds[1].append(pos_courante)
        positions[pos_courante] = (pos_courante, profondeur)
        branches[1].append((pos_courante, pos_parent))

if arbreAsTuple == ():
    return

branches = [[]]
profondeur = hauteur(arbre)
pos_courante = 2 ** profondeur
noeuds = [[pos_courante]]
positions = {pos_courante: (pos_courante, profondeur)}
labels = {pos_courante: str(arbreAsTuple[0])}

if null_node:
    branches.append([])
    noeuds.append([])

profondeur -= 1
parkour(arbreAsTuple[1], noeuds, branches, labels, positions, profondeur,
        pos_courante - 2 ** profondeur, pos_courante, null_node)
parkour(arbreAsTuple[2], noeuds, branches, labels, positions, profondeur,
        pos_courante + 2 ** profondeur, pos_courante, null_node)

mon_arbre = nx.Graph()

if type(size) == int:
    size = (size, size)
plt.figure(figsize=size)
```

Utilisation :

```
from structures import *

a = ABR()
a.setRacine(8)
a.insere(5)
a.insere(3)
a.insere(12)
a.insere(10)
a.insere(15)

afficheArbre(a)
```

Sortie :