TP 9 – Arbres binaires

Année 2019-2020

Ce TP permet de créer des structures de données "arbres binaires" et d'écrire des algorithmes dans le cadre d'applications qui les utilisent. Vous pourrez vous référer à votre cours, ainsi qu'au document `Arbres-TD` sur Moodle fournissant des compléments du cours en vue de l'implémentation.

Exercice 1 – Structure de données `Arbre`

Un arbre est constitué d'un ensemble de noeuds, chaque noeud contenant une valeur. L'arbre binaire comprend un noeud racine et deux sous-arbres gauche et droit donnés respectivement par les attributs `racine`, `gauche` et `droit`. Des méthodes permettent d'accéder à ces attributs, et de tester si l'arbre est vide ou non. On considère des arbres binaires qui sont implémentés avec les classes `Noeud` et `Arbre` données ci-après. Ce code est accessible dans votre espace de travail sur Moodle (programme `arbre`).

```
class Noeud:
    def __init__(self, v):
        self.valeur = v
    def getValeur(self):
        return self.valeur

class Arbre:

    def __init__(self, n, g, d):
        self.racine = n   # noeud racine de l'arbre
        self.gauche = g
        self.droit = d

    def getRacine(self):
        return self.racine

    def valRacine(self):
        return self.racine.getValeur()

    def getGauche(self):
        return self.gauche

    def getDroit(self):
        return self.droit
```

```
def isVide(self):
    return self.racine == None

def isFeuille(self):
    return self.gauche == None and self.droit==None

def hasGauche(self):
    return self.gauche != None;

def hasDroit(self):
    return self.droit != None;
```

1. Créez des noeuds et des arbres et testez les méthodes des classes `Noeud` et `Arbre`. Dans cette implémentation du TAD Arbre, vous considérerez que l'arbre vide est représenté par:

   ```
   aVide = Arbre(None, None, None)
   ```

   Un arbre vide ne peut pas être réduit à `None`, car il n'est pas possible d'appeler une méthode de la classe Arbre à partir de `None`.

2. Écrivez les méthodes de la classe `Arbre` suivantes de manière récursive en n'utilisant que les méthodes données précédemment. Vous implémenterez et testerez ces méthodes sur des arbres à valeurs entières. Les arbres sont binaires, mais les valeurs des noeuds ne sont pas ordonnées.

   - La méthode `hauteur(self)` est donnée. Elle renvoie la hauteur de l'arbre, c'est-à-dire la longueur du plus long chemin descendant de la racine vers une feuille. Testez cette méthode pour des arbres que vous aurez préalablement construits. Modifiez cette méthode `hauteur1(self)` pour changer la convention relative au calcul de la hauteur (cette fois-ci, la hauteur d'une feuille vaut 1 et pas 0).
   - Écrivez la méthode `nb_noeuds(self)` qui renvoie le nombre de noeuds de l'arbre.
   - Écrivez la méthode `somme(self)` qui renvoie la somme de toutes les valeurs contenues dans l'arbre.
   - Écrivez la méthode `incremente(self)` qui renvoie l'arbre dans lequel toutes les valeurs ont été augmentées de 1.

3. **Parcours d'arbres binaires**– Vous considèrerez trois parcours différents :

   - **Parcours préfixe** – Testez la méthode `affiche_prefixe(self)` (donnée dans le fichier `arbre.py`) qui parcourt de manière préfixée l'arbre et affiche la valeur de ses noeuds. Dans ce type de parcours, chaque noeud est visité avant ses fils. En pratique, cela veut dire que vous afficherez d'abord la valeur du noeud courant (en partant initialement de la racine de l'arbre), puis vous appellerez récursivement l'affichage pour le fils gauche puis pour le fils droit du noeud courant. Remarque : il n'existe qu'un seul ordre de parcours de l'arbre respectant cette propriété.
   - **Parcours postfixe** – En vous inspirant de la fonction précédente, écrivez une fonction `affiche_postfixe(self)` qui vérifie la spécification suivante. Un noeud est visité après tous ses descendants et les descendants gauche d'un noeud sont visités avant ses descendants droits (il n'y a ici encore qu'un ordre possible).

- **Parcours infixe** – Écrivez une fonction `affiche_infixe(self)` qui vérifie la spécification suivante. Un noeud est visité après qu'on soit allé dans le descendant gauche. Lorsqu'on a affiché la valeur du noeud, on va alors dans le descendant droit. Dans ce type de parcours, on visite dans l'ordre les feuilles de l'arbre.
- Testez les fonctions de parcours sur des exemples d'arbres contenant des noeuds à valeur entière.

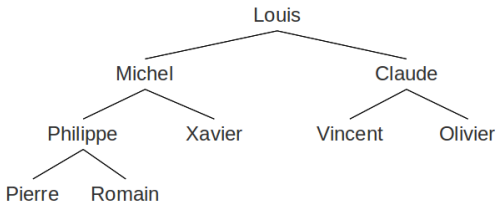
Indication Des exemples de parcours sont donnés dans le cours.

Exercice 2 – Exemples d'utilisation d'arbres binaires

Dans ces exemples vous utiliserez la classe `Arbre` définie à l'exercice 1.

1. **Arbres généalogiques**

   – Écrivez un programme `ArbreGenealogique` qui utilise la classe `Arbre` de l'exercice 1 pour créer un arbre représentant l'arbre généalogique suivant :

   – Complétez le programme précédent pour afficher la liste de tous les petits enfants de Louis (i.e. les éléments de l'arbre à la profondeur 2).

   – Affichez l'arbre généalogique sous la forme suivante :

   ```
   Louis Michel Philippe Pierre Romain Xavier Claude Vincent Olivier
   ```

2. **Calculatrice**

   – Écrivez un programme `Calculatrice` qui utilise la classe `Arbre` pour créer un arbre représentant l'expression suivante : $(1 + 2) \times 3 + (4 \times 5)$

   – Complétez le programme précédent pour qu'il calcule et affiche la hauteur et le nombre de feuilles de l'arbre.

   – Affichez l'expression sous la forme suivante : $1 + 2 \times 3 + 4 \times 5$

   – Complétez le programme `Calculatrice` pour calculer l'expression à l'aide d'une pile en vous aidant de l'exercice réalisé pour le TP sur les piles.

3. **Fonctions récursives avancées** Ecrivez les fonctions récursives suivantes (toujours en utilisant les primitives de manipulation d'arbres.

   – `applique(f,a)` qui applique la fonction f à toutes les valeurs de l'arbre ;

   – `grand_chemin(a)` qui renvoie la plus grande somme possible de valeurs sur un chemin descendant dans l'arbre de la racine à une feuille.