TD: algorithmes sur les arbres binaires:

Dans ce TD, nous allons utiliser deux manières d'implémenter l'arbre binaire suivant:

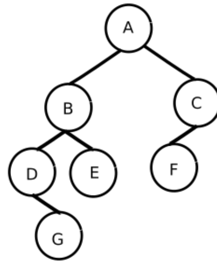

Figure 1: Arbre binaire

```python
#ecriture directe d'un arbre:
arbre1 = ["A",["B",["D",[],["G",[],[]]],["E",[],[]]],["C",["F",[],[]],[]]]


#ecriture d'un arbre sous forme de classe:
class Arbre:
    def __init__(self,val):
        self.valeur = val
        self.gauche = None
        self.droit = None

    def ajout_gauche(self,val):
        self.gauche = Arbre(val)

    def ajout_droit(self,val):
        self.droit = Arbre(val)

arbre2 = Arbre("A")
arbre2.ajout_gauche("B")
arbre2.ajout_droit("C")
arbre2.gauche.ajout_gauche("D")
arbre2.gauche.ajout_droit("E")
arbre2.droit.ajout_gauche("F")
arbre2.gauche.gauche.ajout_droit("G")
```

arbre_binaire_notations.py

Exercice 1: taille d'un arbre binaire:

Il s'agit de déterminer le nombre total de noeuds. Principe de l'algorithme:

- si l'arbre est vide: il y a 0 noeud.
- sinon, on compte le noeud racine plus le nombre de noeuds du sous-arbre gauche et le nombre de noeuds du sous-arbre droit.

Écrire cet algorithme en Python:

1. pour la première implémentation sous forme d'une fonction.
2. pour la seconde implémentation sous forme d'une fonction.
3. pour la seconde implémentation sous forme d'une méthode de la classe Arbre.

Exercice 2: hauteur d'un arbre binaire:
Il s'agit de déterminer le nombre maximal de noeuds se trouvant entre la racine et une feuille. Principe de l'algorithme:

- si l'arbre est vide: il y a 0 noeud.
- sinon, on compte le noeud racine plus le maximum entre le nombre de noeuds du sous-arbre gauche et le nombre de noeuds du sous-arbre droit..

Écrire cet algorithme en Python:

1. pour la première implémentation sous forme d'une fonction.
2. pour la seconde implémentation sous forme d'une fonction.
3. pour la seconde implémentation sous forme d'une méthode de la classe Arbre.

Exercice 3: parcours en profondeur d'abord (DFS pour Depth-First Search) d'un arbre binaire:
On explore chaque branche complètement avant d'explorer la branche voisine. Principe de l'algorithme:

- si l'arbre est non vide, on parcourt de manière récursive son sous-arbre gauche puis son sous-arbre droit.
- sinon, c'est terminé.

On distingue 3 cas suivant le moment où est traité une racine d'un sous-arbre:

- si la racine est traitée avant ses deux sous arbres, il s'agit d'un ordre préfixe.
- si la racine est traitée entre ses deux sous arbres, il s'agit d'un ordre infixe.
- si la racine est traitée après ses deux sous arbres, il s'agit d'un ordre postfixe.

Écrire cet algorithme en Python:

1. pour la première implémentation sous forme d'une fonction.
2. pour la seconde implémentation sous forme d'une fonction.
3. pour la seconde implémentation sous forme d'une méthode de la classe Arbre.

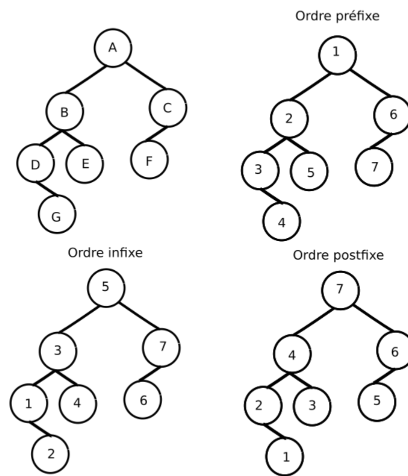

Figure 2: Parcours en profondeur d'un arbre binaire

Exercice 4: parcours en largeur d'abord (BFS pour Breadth-First Search) d'un arbre binaire:
Il s'agit de parcourir un arbre niveau par niveau, en considérant tous les sommets de chaque niveau. On commence par la racine, puis les deux racines de chacun des deux sous-arbres, puis les quatre racines de chacun des quatre sous arbres et ainsi de suite.

La méthode la plus pratique consiste à utiliser une structure de file (deque de la bibliothèque collections en pythons: `https://docs.python.org/fr/3/library/collections.html`) Principe de l'algorithme:

- On place l'arbre dans la file.
- Tant que la file est non vide, on défile un élément (qui est un arbre), on affiche la valeur de la racine et on place dans la file chacun de ses deux sous arbres s'ils ne sont pas vides.

Écrire cet algorithme en Python:

1. pour la première implémentation sous forme d'une fonction.
2. pour la seconde implémentation sous forme d'une fonction.
3. pour la seconde implémentation sous forme d'une méthode de la classe Arbre.

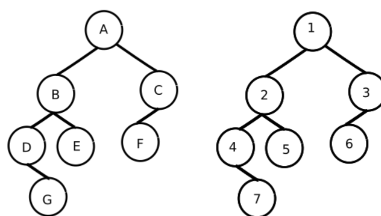

Figure 3: Parcours en largeur d'un arbre binaire