Parcours en largeur

Parcours en largeur d'un arbre binaire

On met en œuvre les files à l'aide de la classe `File` suivante :

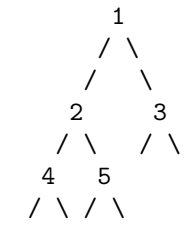
```
from collections import deque

class File:
    def __init__(self):
        self.queue = deque()

    def est_vide(self):
        """Indique si la file est vide"""
        return len(self.queue) == 0

    def enfile(self, x):
        """Enfile x dans la file"""
        self.queue.append(x)

    def defile(self):
        """Défile et renvoie une valeur de la file
        Lève une erreur si la file est vide
        """
        if self.est_vide():
            raise ValueError("La file est vide")
        return self.queue.popleft()
```

Toutes les opérations sur les files se feront par l'intermédiaire des méthodes de cette classe.

Dans cet exercice, on représente les arbres binaires ainsi :

- l'arbre binaire vide est représenté par `None`,
- un arbre binaire non-vide est représenté par le tuple `(<sous-arbre à gauche>, <valeur de la racine>, <sous-arbre à droite>)`.

Ainsi l'arbre



est représenté par `(((None, 4, None), 2, (None, 5, None)), 1, (None, 3, None))`.

Objectif

On demande d'écrire la fonction `parcours_largeur` qui prend en paramètre un arbre binaire représenté comme indiqué ci-dessus et renvoie la liste des valeurs des nœuds rencontrés lors du parcours en largeur issu de sa racine.

On traitera le sous-arbre à gauche avant le sous-arbre à droite.

On rappelle que le parcours en largeur d'un arbre peut s'effectuer à l'aide d'une file.

Exemples

```
>>> arbre_1 = (None, 0, (None, 1, None))
>>> parcours_largeur(arbre_1)
[0, 1]
>>> arbre_2 = ((None, 1, None), 0, (None, 2, None))
>>> parcours_largeur(arbre_2)
[0, 1, 2]
>>> arbre_3 = (((None, 4, None), 2, (None, 5, None)), 1, (None, 3, None))
>>> parcours_largeur(arbre_3)
[1, 2, 3, 4, 5]
```