R3.02 - Structure de données complexes : ARBRES

Un **arbre** est une structure de données non linéaire composée de nœuds, où chaque nœud contient une donnée et un ensemble de nœuds enfants.

Contrairement aux structures comme les listes ou les tableaux, **un arbre n'est pas séquentiel**.

Caractéristiques d'un arbre :

- **Nœud** : un élément de l'arbre qui contient des données.
- **Racine** : le nœud de départ de l'arbre. Un arbre a une seule racine.
- **Enfant** : un nœud relié à un autre nœud qui est considéré comme le parent.
- **Parent** : un nœud qui a des enfants. Tout nœud (excepté la racine) a un parent.
- **Feuille** : un nœud qui n'a pas d'enfants.
- **Hauteur de l'arbre** : la longueur maximale du chemin de la racine à une feuille.
- **Profondeur d'un nœud** : le nombre d'arêtes entre ce nœud et la racine.
- **Sous-arbre** : toute partie de l'arbre qui peut être considérée comme un arbre en elle-même, à partir d'un nœud donné.

Types d'arbres

Arbre binaire

Un arbre dans lequel chaque nœud a au plus deux enfants.

Arbre n-aire

Un arbre où chaque nœud peut avoir jusqu'à *n* enfants.

Arbre de recherche binaire (BST)

Un arbre binaire où pour chaque nœud, tous les enfants du sous-arbre gauche ont des valeurs inférieures à la valeur du nœud et ceux du sous-arbre droit ont des valeurs supérieures.

Arbre équilibré

Un arbre où la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit de chaque nœud est au plus 1.

Arbre AVL, B-Tree, etc.

D'autres types spécifiques d'arbres utilisés dans diverses applications.

Implémentation générale d'un arbre

```
class Noeud:
    def __init__(self, valeur):
        self.valeur = valeur  # donnée du nœud
        self.enfants = []     # liste des enfants du nœud
```

```
    def ajouter_enfant(self, enfant):
        self.enfants.append(enfant)  # ajouter un enfant à la liste des
enfants

class Arbre:
    def __init__(self):
        self.racine = None  # l'arbre commence sans racine

    def afficher_arbre(self, noeud, niveau=0):
        if noeud is not None:
            print(" " * niveau * 2 + str(noeud.valeur))  # indentation
selon le niveau
            for enfant in noeud.enfants:
                self.afficher_arbre(enfant, niveau + 1)

# ---------- Exemple d'utilisation -------
arbre = Arbre()

# Création des nœuds
racine = Noeud("Racine")
enfant_1 = Noeud("Enfant 1")
enfant_2 = Noeud("Enfant 2")
petit_enfant_1 = Noeud("Petit Enfant 1")
petit_enfant_2 = Noeud("Petit Enfant 2")

# Construction de l'arbre
racine.ajouter_enfant(enfant_1)
racine.ajouter_enfant(enfant_2)
enfant_1.ajouter_enfant(petit_enfant_1)
enfant_2.ajouter_enfant(petit_enfant_2)

# Assigner la racine à l'arbre
arbre.racine = racine

# Affichage de l'arbre
arbre.afficher_arbre(arbre.racine)
```

Résultat

```
Racine
  Enfant 1
    Petit Enfant 1
  Enfant 2
    Petit Enfant 2
```

Questions de compréhension

- Quelle est la hauteur de cet arbre ?
- Est-il équilibré ?
- Quelles sont ses feuilles ?

Applications des arbres

Les arbres sont utilisés dans plusieurs domaines de l'informatique :

Systèmes de fichiers

L'organisation des fichiers et des répertoires dans un système d'exploitation est souvent représentée sous forme d'arbre.

Analyse syntaxique

En compilation, les expressions sont souvent représentées comme des arbres syntaxiques.

Intelligence artificielle

Les arbres de décision sont utilisés pour prendre des décisions en fonction de plusieurs conditions.

Bases de données

Des arbres comme les B-Trees sont utilisés pour organiser les données dans les systèmes de gestion de bases de données.

Exemple du tri grâce à un arbre binaire

```
class Noeud:
    def __init__(self, valeur):
        self.valeur = valeur  # La donnée du nœud
        self.gauche = None    # Enfant gauche
        self.droite = None    # Enfant droit

class ArbreBinaire:
    def __init__(self):
        self.racine = None  # Le premier nœud de l'arbre (la racine)

    def ajouter(self, valeur):
        # Si l'arbre est vide, le nouveau nœud devient la racine
        if self.racine is None:
            self.racine = Noeud(valeur)
        else:
            self._ajouter_recursivement(self.racine, valeur)

    def _ajouter_recursivement(self, noeud, valeur):
        # Si la valeur est plus petite, on va à gauche
        if valeur < noeud.valeur:
            if noeud.gauche is None:
```

```
                noeud.gauche = Noeud(valeur)
            else:
                self._ajouter_recursivement(noeud.gauche, valeur)
        # Si la valeur est plus grande, on va à droite
        else:
            if noeud.droite is None:
                noeud.droite = Noeud(valeur)
            else:
                self._ajouter_recursivement(noeud.droite, valeur)

    def afficher_en_ordre_avec_indentation(self, noeud=None, niveau=0):
        if noeud is None:
            noeud = self.racine
        # Affiche le sous-arbre gauche
        if noeud.gauche is not None:
            self.afficher_en_ordre_avec_indentation(noeud.gauche, niveau + 1)
        # Affiche la valeur du nœud avec l'indentation selon son niveau
        print(" " * (niveau * 2) + str(noeud.valeur))
        # Affiche le sous-arbre droit
        if noeud.droite is not None:
            self.afficher_en_ordre_avec_indentation(noeud.droite, niveau + 1)

# Utilisation
arbre = ArbreBinaire()
arbre .ajouter(10)
arbre .ajouter(5)
arbre .ajouter(15)
arbre .ajouter(3)
arbre .ajouter(7)
arbre .ajouter(8)
arbre .ajouter(12)
arbre .ajouter(1)
arbre .ajouter(6)
arbre .ajouter(20)

# Affichage avec indentations
arbre.afficher_en_ordre_avec_indentation()
```

Résultat

```
      1
    3
  5
      6
    7
      8
10
    12
  15
    20
```

Questions de compréhension

- Quelle est la hauteur de cet arbre ?
- Est-il équilibré ?
- Quelles sont ses feuilles ?
- rédiger une fonction afficher en ordre (sans indentation)