Type de document	Résumé de cours	Classe	Tle	Durée	2h	Date	18/11/2024
Thème et contenu(s)	Structures de données : arbres binaires						
Capacités attendues	Connaître le vocabulaire de base, implémenter une structure arborescente en POO						
Prérequis	Objets en Python						
Description	Résumé du cours sur les arbres : définition et implémentation						

I) Les arbres : définitions

a) Définition générale

Les **arbres** sont des **structures de données hiérarchiques** (par opposition aux **structures de données linéaires** comme les listes à 1 dimension), très utilisées en informatique. Ils permettent par exemple, comme leur nom l'indique, de décrire une **arborescence**, de dossiers/fichiers ou autres (contenant-contenu). En terminale, nous nous intéressons aux **arbres binaires**.

b) Nœud et clef

Un arbre est constitué de **nœuds**. Le nœud de plus haut niveau (niveau zéro) s'appelle la **racine** : la racine est le seul nœud de l'arbre qui n'a aucun parent. Un **nœud** (autre que la racine) a **un seul nœud parent** mais peut avoir **plusieurs enfants**. Un nœud peut contenir une valeur (numérique, textuelle…etc), qu'on appelle souvent sa **clef**.

c) Feuille

Un nœud sans enfants est appelé **feuille**.

d) Profondeur et hauteur

La **profondeur d'un nœud** est sa distance à la racine de l'arbre (son niveau). La **hauteur d'un arbre** est donnée par la **distance** entre son **nœud le plus profond et la racine**. Autrement dit, la hauteur d'un arbre est la « profondeur de son nœud le plus profond ». On parle donc de profondeur pour un nœud, et de hauteur pour un arbre. Pour la hauteur les avis divergent : pour moi (et d'autres) un arbre composé d'un seul nœud (sa racine donc) est de hauteur 0, pour Eduscol il est de hauteur 1.

e) Arbre binaire

Un **arbre binaire** est un arbre dont chaque **nœud** a **au plus 2 fils**, que l'on appelle souvent par commodité **fils gauche** et **fils droit**. En d'autres termes, un nœud possède zéro, un ou deux fils.

f) Arbre binaire strict (localement complet)

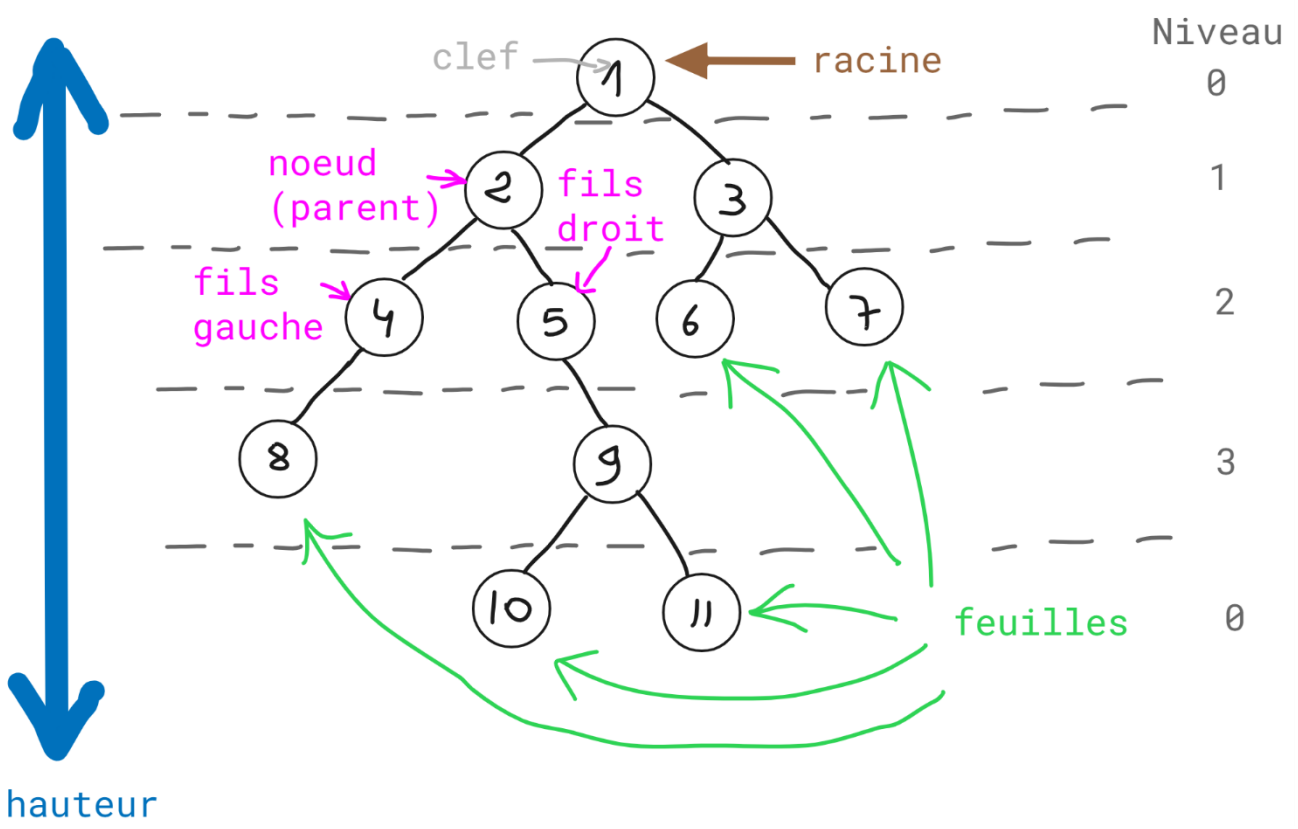
C'est un **arbre binaire** dont les **nœuds** comportent **zéro ou deux fils**.

g) Arbre binaire parfait

C'est un arbre binaire strict (localement complet) dont **toutes les feuilles sont de même profondeur** (i.e. toutes les feuilles ont la même distance à la racine est la même). **Tous les niveaux d'un arbre binaire parfait sont « remplis »**. Il y a une relation évidente entre les puissances de deux et les niveaux : si n est le niveau, un arbre binaire parfait comporte 2^n feuilles à ce niveau.

II) Les arbres : représentation

Vous pouvez retrouver ici les notions du point précédent représentées graphiquement.



III) Implémentation

Commençons par implémenter les **arbres binaires** en utilisant la POO qui va nous faciliter la tâche. Vous devez revoir le cours sur les objets : rappelez-vous qu'un attribut, dans une classe, peut être une valeur primitive ('bleu', 2.58, True) ou…un autre objet.
Comme dans toute implémentation un tant soit peu complexe, il existe plusieurs variantes.

a) Idée de base

Elles partagent toutefois la même idée : **un arbre est un pointeur vers son premier élément** (sa racine). On « navigue » ensuite dans l'arbre grâce à des **primitives**, qui sont en fait des **méthodes**. Ces primitives constituent **l'interface** qui nous permet de manipuler un arbre. C'est exactement le même principe que pour les Piles et les Files, sauf que pour ces dernières nous avons utilisé le **procédural** et les **fonctions**, et que là nous utiliserons **l'objet** et les **méthodes**.

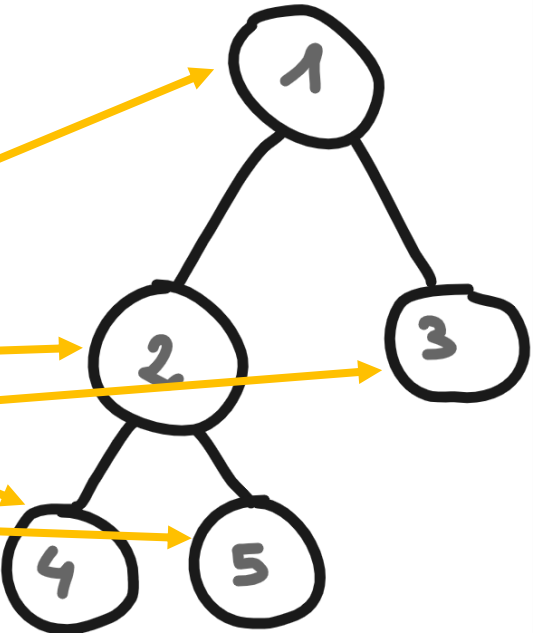
b) Implémentation simple

Voici un exemple d'implémentation simple.

```
1   class Noeud:
2
3       def __init__(self, clef):
4           self.gauche = None #pointe vers le fils gauche
5           self.droit = None #pointe vers le fils droit
6           self.clef = clef #contient la clef (valeur littérale)
```

On peut, même sans créer d'autres méthodes, peupler notre arbre, même si cela reste peu pratique…voyez par exemple une implémentation (à gauche) d'un **arbre binaire strict** (à droite)

```
monArbre = Noeud(1)
monArbre.gauche = Noeud(2)
monArbre.droit = Noeud(3)
monArbre.gauche.gauche = Noeud(4)
monArbre.gauche.droit = Noeud(5)
```



Ici, on est partis de la racine pour ajouter des éléments plus en profondeur. Remarquez la notation faite d'une succession de « . ».

On pourrait faire exactement pareil en démarrant par les feuilles de l'arbre :

```
noeud4 = Noeud(4) #feuille 4
noeud5 = Noeud(5) #feuille 5
noeud2 = Noeud(2)  #noeud 2
noeud2.gauche = noeud4 #ajout de la feuille 4 (fils gauche) au noeud 2
noeud2.droit = noeud5 #ajout de la feuille 5 (fils droit) au noeud 2
noeud3 = Noeud(3) #le noeud 3 n'a pas d'enfants
noeud1 = Noeud(1) #création de la racine
noeud1.gauche = noeud2 #ajout du fils gauche
noeud1.droit = noeud3 #ajout du fils droit
```

c) Calcul de la hauteur : avec une fonction

Comme nous le dit la définition de la **hauteur** d'un arbre, il s'agit de la distance maximale entre sa feuille de plus grande profondeur et sa racine.

On peut calculer la hauteur d'un arbre binaire en utilisant la récursivité : pour chaque nœud, la hauteur est de 1 + la hauteur maximale (choisie entre le fils gauche et le fils droit).

Il faut penser au cas où le nœud est une feuille, et pour cela on peut écrire une méthode simple (une feuille est un nœud qui ne possède ni enfant gauche, ni enfant droit) :

```
def estFeuille(self):
    if self.gauche == None and self.droit == None:
        return True
    else:
        return False
```

Passons maintenant au calcul de la hauteur. Le plus simple, pour commencer, est de ne plus utiliser les méthodes mais les fonctions, c'est certes un peu déroutant mais c'est pour que vous compreniez mieux. Dans la littérature, énormément d'exemples calculent la hauteur d'un arbre à partir d'une fonction externe à la classe, et non pas à partir d'une méthode de la classe Arbre ou Nœud. Donc on se met à l'extérieur de la classe et on code ainsi :

```
#c'est une fonction pas une méthode !
def hauteur(n):
    if n.estFeuille():
        return 0
    else:
        return 1 + max(hauteur(n.gauche),hauteur(n.droit))
```

On peut la tester sur nos deux arbres :

```
print(hauteur(monArbre))
print(hauteur(noeud1))
```

Sans surprise, le programme renvoie 2 à chaque fois. On peut encore simplifier l'algorithme suivant et ne pas utiliser de méthode pour vérifier si le nœud est une feuille. En effet, lorsqu'on appelle récursivement la fonction hauteur, elle prend toujours en paramètre un nœud : celui-ci peut être None, c'est cela que nous allons utiliser pour symboliser qu'il n'existe pas.

```python
#c'est une fonction pas une méthode !
def hauteur(n):
    if n is None:
        return 0
    else:
        return 1 + max(hauteur(n.gauche),hauteur(n.droit))
```

Cette manière de raisonner nous conduit d'ailleurs à avoir, pour nos deux arbres d'exemple, une hauteur de 3. Pour que la fonction précédente fasse de même, on aurait pu écrire :

```python
#c'est une fonction pas une méthode !
def hauteur(n):
    if n.estFeuille():
        return 1
    else:
        return 1 + max(hauteur(n.gauche),hauteur(n.droit))
```

Vous avez désormais compris comment utiliser la récursivité pour calculer la hauteur d'un arbre.

Réflexion : Pourquoi est-ce déroutant ? On a envie d'utiliser notre classe, certes, mais…le mieux pour examiner un objet dans sa totalité (compter des valeurs, le parcourir…) est parfois d'utiliser un élément externe à cet objet. Maintenant que vous avez compris le principe, on peut tout à fait calculer la hauteur de l'arbre grâce à une méthode

d) Calcul de la hauteur : avec une méthode

On doit prendre soin à une chose : il ne faut pas appeler systématiquement la méthode `hauteur()` sur les fils gauches et droits. En effet, l'un des deux (ou les deux) peuvent valoir `None`, et l'objet `None` ne possède pas de méthode `hauteur()`
Une fois ce cas pris en compte, on peut tout à fait calculer la hauteur d'un arbre au moyen d'une méthode :

```python
def hauteur(self):

    if self.gauche is None:
        fg = 0
    else:
        fg = self.gauche.hauteur()

    if self.droit is None:
        fd = 0
    else:
        fd = self.droit.hauteur()

    return 1+max(fg,fd)
```

e) Calcul de la taille

L'algorithme de calcul de la taille est semblable à celui qui calcule la hauteur de l'arbre.

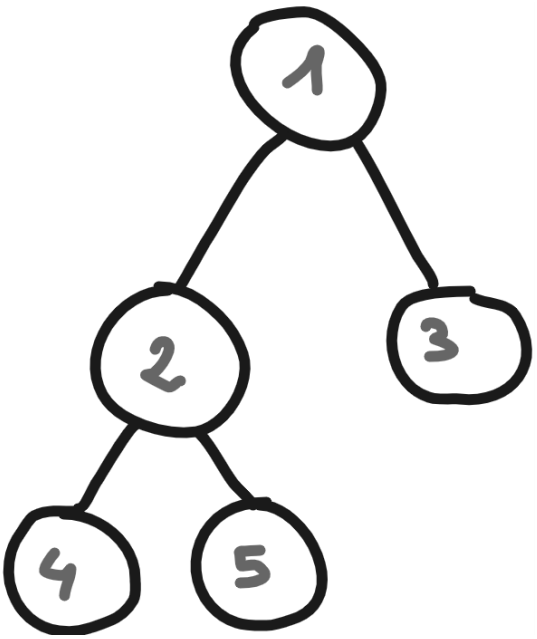
```
def taille(self):

    if self.gauche is None:
        fg = 0
    else:
        fg = self.gauche.taille()

    if self.droit is None:
        fd = 0
    else:
        fd = self.droit.taille()

    return 1 + fg + fd
```

IV) Algorithmes de parcours

Il existe deux grandes familles de parcours :

- le **parcours en largeur** (BFS, Breadth-First Search), qui consiste à parcourir chaque niveau de l'arbre avant de passer au niveau suivant
- le **parcours en profondeur** (DFS, Depth-First Search), qui consiste à parcourir chaque branche avant de passer à la branche suivante. Il se compose lui-même de 3 types de parcours :
    - si on traite la racine avant ses deux sous-arbres : c'est un **parcours préfixe**
    - si on traite la racine après ses deux sous-arbres : c'est un **parcours postfixe**
    - si on traite la racine après le sous-arbre gauche mais avant le sous-arbre droit : c'est un **parcours infixe**

Pour l'arbre suivant :



- Le parcours BFS donne : 1,2,3,4,5
- Le parcours préfixe donne : 1,2,4,5,3
- Le parcours infixe donne : 4,2,5,1,3
- Le parcours postfixe donne : 4,5,2,3,1