Cours 17 décembre — Résumé de la séance

Nous avons continué le travail sur les arbres binaires. Je vous rappelle qu'un arbre binaire est un ensemble de Noeuds ayant 3 attributs : une valeur, un sous-arbre gauche et un sous-arbre droit. Le sous-arbre gauche (ou droit) sera lui aussi un ensemble de Noeuds. On a donc une structure récursive.

Comme indiqué dans le premier résumé de cours, nous avons continué le TP sur les arbres binaires. Je vous fournis ici une correction détaillée des différents exercices.

Nous avons ensuite travaillé sur l'exercice 3 permettant de générer des arbres parfaits de hauteur donnée.

Finalement, pour les vacances de Noel, je vous donne un petit DM (c'est l'exercice 5 à faire pour la rentrée) ainsi que la fin du cours à lire (jusqu'à la partie Encapsulation).
Afin de visualiser les arbres, je vous propose un module disponible sur mon site web. Celui-ci s'appelle GVizProto.py et contient **BEAUCOUP** de méthodes, fonctions et autres. Pas la peine de regarder trop en détail comment il fonctionne ! Mais, si vous le faites, peut-être trouverez-vous le sujet su BAC blanc caché quelque part ?

Pour utiliser ce module, vous le lancez et vous avez une invite de commande en bas. Les arbres s'écrivent de la même manière que celle définie dans le DM :

1 noeud nommé A : [A]
1 racine nommé A et 1 sous-arbre droit nommé B : [A[B]]
1 racine nommé A, 1 sous-arbre droit nommé B qui a deux fils : [A[[C]B[D]]]

Essayez, c'est interactif. Il y a encore probablement des bugs. Trouvez-les et faites-m'en part 😇

1) Arbres binaires — TP — 1 heure 30

Nous avons commencé la séance par le TP les arbres binaires :

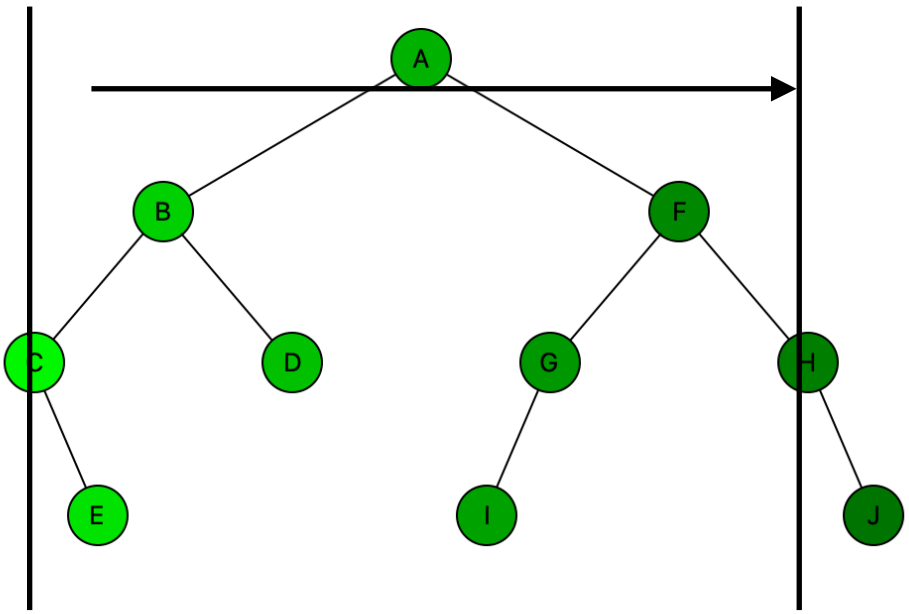
Parcours infixe (exercice 4 du TP)
L'idée du parcours infixe est de lire l'arbre comme si on le parcourait de gauche à droite à l'aide d'une règle. C'est d'ailleurs la méthode que vous devez utiliser si on vous pose cette question au BAC.

a) Cette fonction est clairement écrit en Orienté Objet. Attention, la "récursivité" n'est pas un paradigme. On va regrouper toutes les méthodes concernant les arbres dans une même boite à outils : le parcours infixe est une de ces méthodes.
b) Non, clairement :

```
    def parcoursInfixe(self, a):
        if a is None: return
        self.parcoursInfixe(a.gauche)
        print(a.valeur)
        self.parcoursInfixe(a.droit)
```

c) parcoursInfixe s'appelle lui-même. Les lignes 2 et 3 concernent la condition d'arrêt : on voit qu'on s'arrête quand l'Arbre est Vide (et non que le sous-arbre est vide). Cela permet de traiter le cas où quelqu'un souhaiterait appeler parcours infixe sur l'arbre vide.

d) On lit d'abord les arbres gauches, la racine et ensuite les arbres droits. Cela donne.




```
C   E   B   D   A   I  G     F     H  J
```

Parcours en largeur (exercice 5 du TP)

L'idée du parcours en largeur est de parcourir les noeuds par ordre de hauteur croissante. On va utiliser une file pour faire cela. On défilera le premier élément de la file et on enfilera en queue de file les deux fils de ce Noeud. Ainsi, dans la file à une hauteur k donné, on a toujours une structure: [hauteur k, hauteur k …. hauteur k, hauteur k+1 …. hauteur k+1]

a) Cette structure de données est une file

b) Ⓑ représentera l'arbre B alors que B représente la valeur.

On se fait un tableau pour résumer la situation :

f = [Ⓐ]

x	f	affichage	Tg	f	Td	f
Ⓐ	[]	A	Ⓑ	[Ⓑ]	Ⓕ	[Ⓑ, Ⓕ]
Ⓑ	[Ⓕ]	B	Ⓒ	[Ⓕ, Ⓒ]	Ⓓ	[Ⓕ, Ⓒ, Ⓓ]
Ⓕ	[Ⓒ, Ⓓ]	F	Ⓖ	[Ⓒ, Ⓓ, Ⓖ]	Ⓗ	[Ⓒ, Ⓓ, Ⓖ, Ⓗ]
Ⓒ	[Ⓓ, Ⓖ, Ⓗ]	C	None	[Ⓓ, Ⓖ, Ⓗ]	Ⓔ	[Ⓓ, Ⓖ, Ⓗ, Ⓔ]
		D				
		G				
		H	puis E, I et J			

c) voir présentation de l'algorithme.
d) Le plus complexe ici est de comprendre ce qu'est la racine d'un arbre. Pour nous, il s'agit en fait d'un Noeud, car nous n'avons pas encore encapsulé nos Noeuds dans un objet Arbre. Nous devons donc simplement écrire :

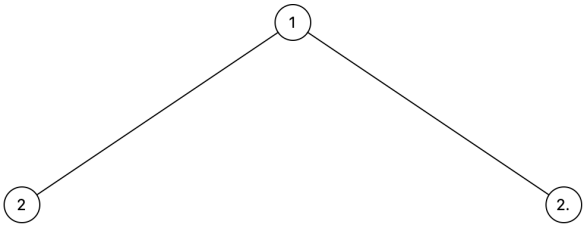
```
def parcoursLargeurEleve(G: Noeud):
    print("DEBUT")
    f = []
    f.append(G)
    while len(f) != 0:
        x = f.pop(0)
        print(x.valeur)
        if x.gauche != None:
            Tg = x.gauche
            f.append(Tg)
        if x.droit != None:
            Td = x.droit
            f.append(Td)
        print(x, x.valeur, x.gauche, x.droit, f) # affichage pour
voir
```

5) Un arbre parfait : exercice 3 de la feuille d'ex — 30 minutes

Comment faire un arbre parfait ? On pourrait imaginer bidouiller une fonction avec des files. Ce ne serait sans doute pas très très joli.

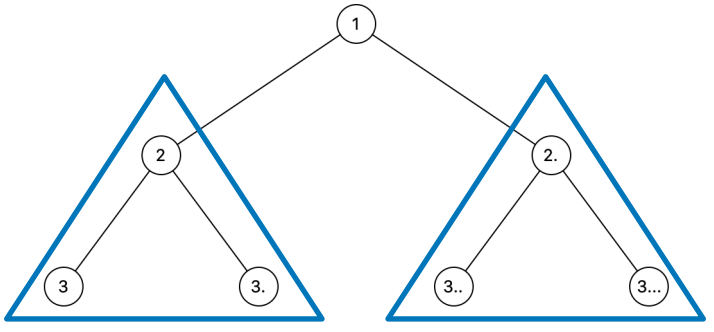
Alors qu'avec une procédure récursive, tout devient lisible !

hauteur 1 : l'arbre à un noeud est parfait.
hauteur 2 : Un arbre à trois noeuds est parfait aussi : mais de quoi est-il constitué ?? Il est constitué d'une racine, d'un sous-arbre gauche à un noeud et d'un sous-arbre droit à un noeud.



Mouais… Elle est où la récursivité ? Continuons

hauteur 3 : Un arbre à cinq noeuds est encore parfait : mais de quoi est-il constitué ?? Il est encore une fois constitué d'une racine, d'un sous-arbre gauche à 3 noeuds et d'un sous-arbre droit à trois noeuds.



Bon, ca nous avance pas : pour comprendre la récursivité, il faut trouver une propriété qui revient sur une version plus petite du problème. Clairement le nombre de noeuds, c'est pas bon.

Reformulons donc :
Mon arbre parfait de **hauteur 3** est donc constitué **d'une racine**, d'un sous-arbre gauche **parfait** de **hauteur 2** et d'un sous-arbre droit **parfait** de **hauteur 2**.

Ca a l'air plus vendeur ! Si on a bien compris :
Mon arbre parfait de **hauteur k** est donc constitué **d'une racine**, d'un sous-arbre gauche **parfait** de **hauteur k-1** et d'un sous-arbre droit **parfait** de **hauteur k-1**.

Allez, c'est parti pour le code :

```
def parfait(h):
    if h == 0 : return None
    return Noeud(parfait(h-1), h, parfait(h-1))
```

Pour le cas terminal, on aurait aussi pu écrire tout en sachant qu'on laisse le cas h=0 de côté (arbre parfait vide, mouais)....

```
    if h == 1 : return Noeud(None, h, None)
```