

Les arbres binaires

Introduction

Parmi la forêt d'arbres possibles, on s'intéressera essentiellement aux arbres dit binaires:

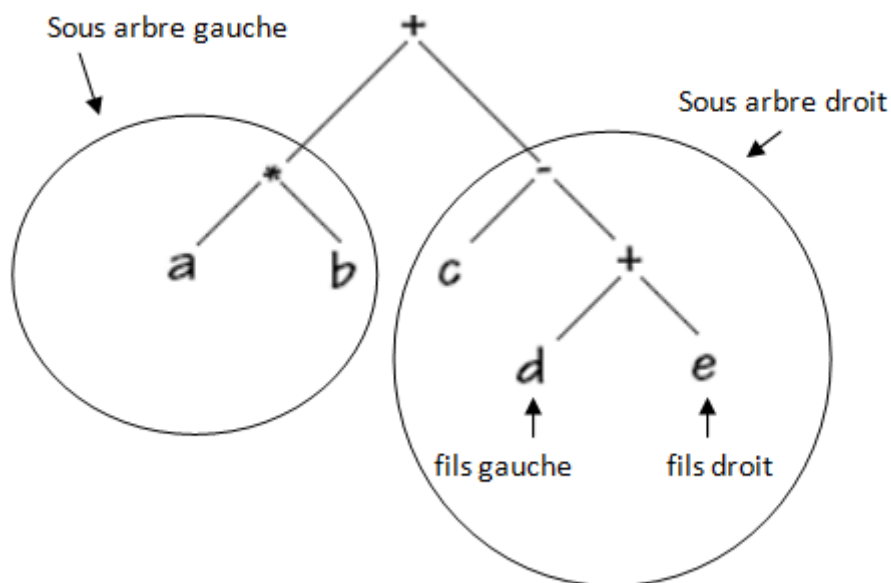
DÉFINITION :

Un **arbre binaire** est un **arbre de degré 2** (dont les noeuds sont de degré 2 au plus).

Vocabulaire :

Les enfants d'un noeud sont lus de **gauche à droite** et sont appelés: **fils gauche et fils droit**.

L'arbre qui représente l'expression $a \times b + c - (d + e)$ est un arbre binaire



QUESTION 1:

Parmi les arbres , du cours sur les arbres, lesquels sont binaires?

.....
.....

REMARQUE :

Les arbres binaires forment une structure de données qui peut se définir de façon **récur-sive**.

Un arbre binaire est :

- Soit **vide**.
- Soit composé d'une **racine portant une étiquette** (clé) et d'une **paire d'arbres binaires**, appelés **sous-arbres gauche et droit**.

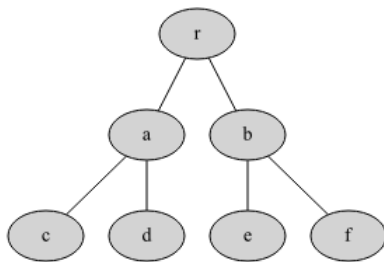
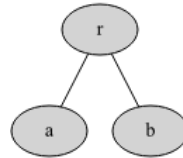
Les arbres binaires sont utilisés dans de très nombreuses activités informatiques, nous allons étudier et implémenter cette structure de données.

Comment représenter un arbre binaire...à la main...

L'idée est de représenter l'arbre avec un tableau:

r	a	b
---	---	---

 La racine suivie de ses fils gauche et droit.



Puis on rajoute dans l'ordre les fils gauche et droit de a, puis ceux de b.

r	a	b	c	d	e	f
---	---	---	---	---	---	---

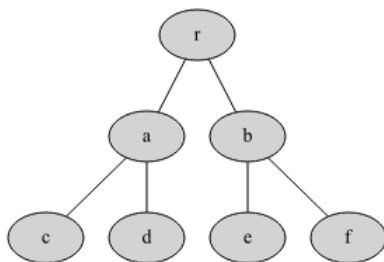
REMARQUE :

Chaque **noeud** se repère par **son indice n dans la liste**, son **fils gauche** se trouvant alors à l'**indice $2n + 1$** et son **fils droit** à l'**indice $2n + 2$** . **r** est à l'indice 0
b est à l'indice 2, son fils gauche se trouve alors à l'indice 5 et son fils droit à l'indice 6.

Si un noeud n'a pas de fils on le précise en mettant 'None' à sa place. Notre arbre est alors représenté par le tableau :

[r, a, b, c, d, e, f, **None**, **None**, **None**, **None**, **None**, **None**, **None**, **None**]

Quelle taille doit avoir le tableau?



Cet arbre est complet (tous les noeuds internes ont deux fils).

Il possède 3 niveaux, sa hauteur ou profondeur est donc de 2

Donc la taille du tableau sera de :

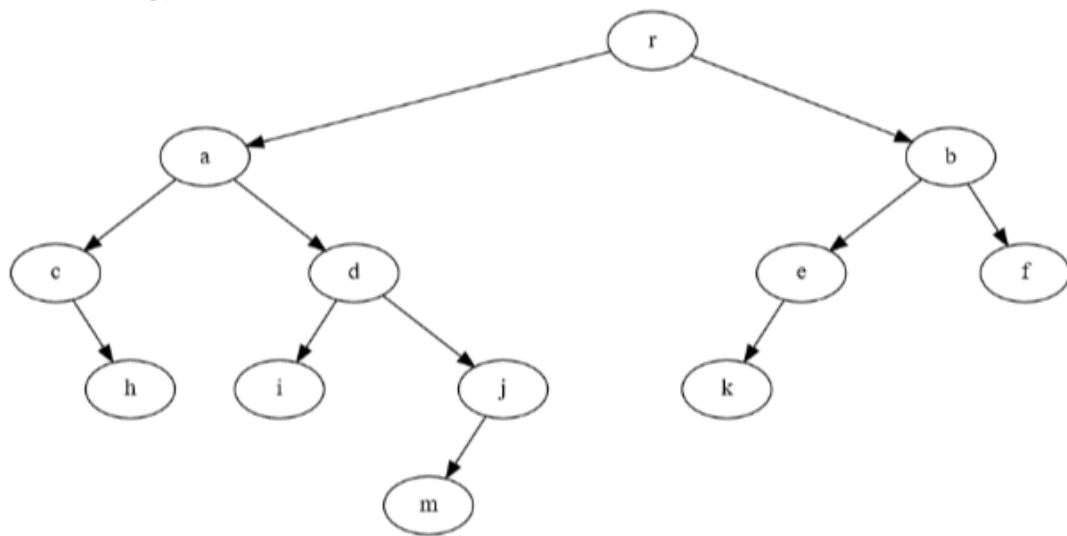
$$2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1 = 15$$

il faut compter le nombre de noeuds, y compris les noeuds "fantômes" des feuilles

La taille correspond au nombre de noeux de l'arbre, y compris les noeuds fantômes.

EXERCICE 1 :

Quelle est la taille du tableau qui permet de représenter cet arbre? :



.....

Écrire le tableau représentant cet arbre :

.....
.....
.....
.....
.....

QUESTION 2:

Quelle propriété ont les indices des fils gauches et droits?

.....
.....
.....

QUESTION 3:

Voici un tableau représentant un arbre binaire :

['*', '-', 5, 2, 6, **None, None, None, None, None, None, None, None, None**]

Le dessiner

Que peut il représenter?

.....
.....
.....
.....
.....
.....

Voici un code Python qui crée la liste représentant l'arbre de l'exercice 1 :

Juste pour ne pas avoir à l'écrire à la main...En Python un tableau est une liste...

```
def creation_arbre(r,profondeur):
    ''' r : la racine (str ou int). la profondeur de l'arbre (int)'''
    Arbre = [r]+[None for i in range(2**(profondeur+1)-2)]
    return Arbre

def insertion_noeud(arbre,n,fg,fd):
    '''Insère les noeuds et leurs enfants dans l'arbre'''
    indice = arbre.index(n)
    arbre[2*indice+1] = fg
    arbre[2*indice+2] = fd
# création de l'arbre
arbre = creation_arbre("r",5)
# ajout des noeuds par niveau de gauche à droite
insertion_noeud(arbre,"r","a","b")
insertion_noeud(arbre,"a","c","d")
insertion_noeud(arbre,"b","e","f")
insertion_noeud(arbre,"c",None,"h")
insertion_noeud(arbre,"d","i","j")
insertion_noeud(arbre,"e","k",None)
insertion_noeud(arbre,"f",None,None)
insertion_noeud(arbre,"h",None,None)
insertion_noeud(arbre,"i",None,None)
insertion_noeud(arbre,"j","m",None)
insertion_noeud(arbre,"k",None,None)
insertion_noeud(arbre,"m",None,None)
#pour vérifier
print(len(arbre))
print(arbre)
```

Voici une fonction qui retourne le parent d'un noeud s'il est dans l'arbre:

```
def parent(arbre,p):
    if p in arbre:
        indice = arbre.index(p)
        if indice%2 == 0:
            return arbre[(indice-2)//2]
        else:
            return arbre[(indice-1)//2]
```

À FAIRE 1:

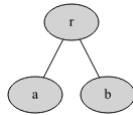
Créer les fonctions suivantes :

- Une fonction qui retourne vrai si l'arbre est vide.
- Une fonction qui retourne les enfants d'un noeud.
- Deux fonctions qui retournent le fils gauche d'un noeud et son homologue le fils droit s'ils existent.
- Une fonction qui retourne vrai si le noeud est la racine de l'arbre.
- Une fonction qui retourne vrai si le noeud est une feuille.
- Une fonction qui retourne vrai si le noeud a un frère gauche ou droit.

Une seconde façon de faire...

Comme vous l'avez sans doute constaté, il est assez fastidieux de représenter un arbre avec un unique tableau surtout pour un arbre très profond.

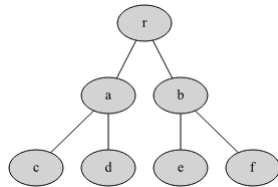
L'idée est de représenter l'arbre avec un tableau contenant des tableaux



Cet arbre se représente par le tableau:
['r', ['a', [], []], ['b', [], []]]

EXERCICE 2 :

Écrire le tableau représentant cet arbre :



.....
.....
.....
.....
.....

Le code Python ci-dessous, construit l'arbre de l'exercice 1 de manière récursive.

- Les noeuds sont représentés par un dictionnaire.
- L'arbre se construit depuis la racine en construisant les sous-arbres des fils gauche et droit.

```
def noeud(nom, fg = None, fd = None) :  
    return {'racine': nom, 'fg' : fg, 'fd': fd}  
# création des noeuds  
k = noeud('k')  
f = noeud('f')  
e = noeud('e', k, None)  
b = noeud('b', e, f)  
m = noeud('m')  
j = noeud('j', m, None)  
i = noeud('i')  
d = noeud('d', i, j)  
h = noeud('h')  
c = noeud('c', None, h)  
a = noeud('a', c, d)  
racine = noeud('r', a, b)  
# création de l'arbre  
def construit(arbre) :  
    if arbre == None :  
        return []  
    else:  
        return [arbre['racine'], construit(arbre['fg']), construit(arbre['fd'])]  
arbre1=construit(racine)  
print(arbre1)
```

Les fonctions du "À faire 1" sont assez évidentes à écrire puisqu'on a accès directement aux noeuds.

Nous allons nous intéresser à la hauteur de l'arbre, ainsi qu'aux différentes façons de le parcourir:

- Le parcours en largeur.
- Les parcours en profondeur (parcours préfixe, parcours infixe et parcours suffixe).

Calcul de la hauteur de l'arbre :

L'idée est la suivante :

- Si l'arbre est vide la hauteur vaut -1.
- Sinon la hauteur vaut 1 auquel il faut ajouter le maximum entre les hauteurs des sous-arbres gauche et droit.
- Ces sous-arbres sont eux mêmes des arbres dont il faut calculer la hauteur.

Une méthode récursive semble tout à fait adaptée à la situation.

Voici l'algorithme

```
Fonction hauteur (arbre)
Données : arbre binaire
représenté comme tableau de tableaux
[racine , [fg] , [fd] ]
et fg=[noeud , [fg] , [fd] ]

Si l'arbre est vide alors
  | renvoyer -1
Sinon
  | h1←1+hauteur(arbre[1])
  | h2←1+hauteur(arbre[2])
  | renvoyer max(h1,h2)
```

À FAIRE 2:

Écrire cette fonction dans le code précédent et vérifier que la profondeur de l'arbre est de 4.

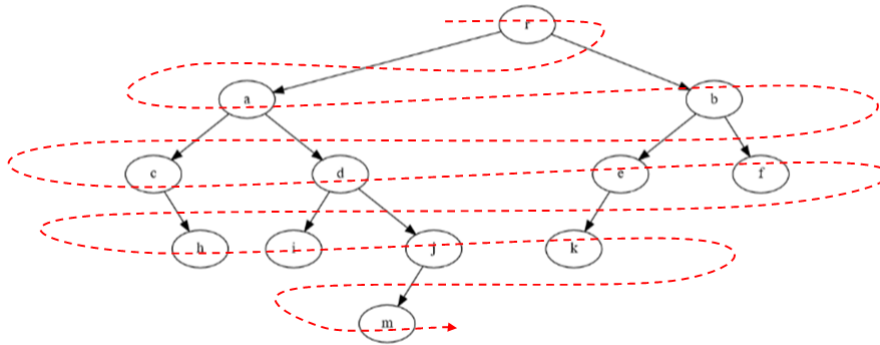
EXERCICE 3 :

Dans un nouveau fichier, implémenter l'arbre de la question 3 et faites calculer sa profondeur

Les parcours

Le parcours en largeur :

Le parcours d'un arbre en largeur consiste à partir de la racine on visite ensuite son fils gauche puis son fils droit, puis le fils gauche du fils gauche etc.. Comme le montre le schéma ci-dessous:



L'idée est la suivante:

On utilise une File.

- On met l'arbre dans la file.
- Puis tant que la file n'est pas vide:
 - On défile la file.
 - On récupère sa racine.
 - On enfile son fils gauche s'il existe.
 - On enfile son fils droit s'il existe.

Voici l'algorithme:

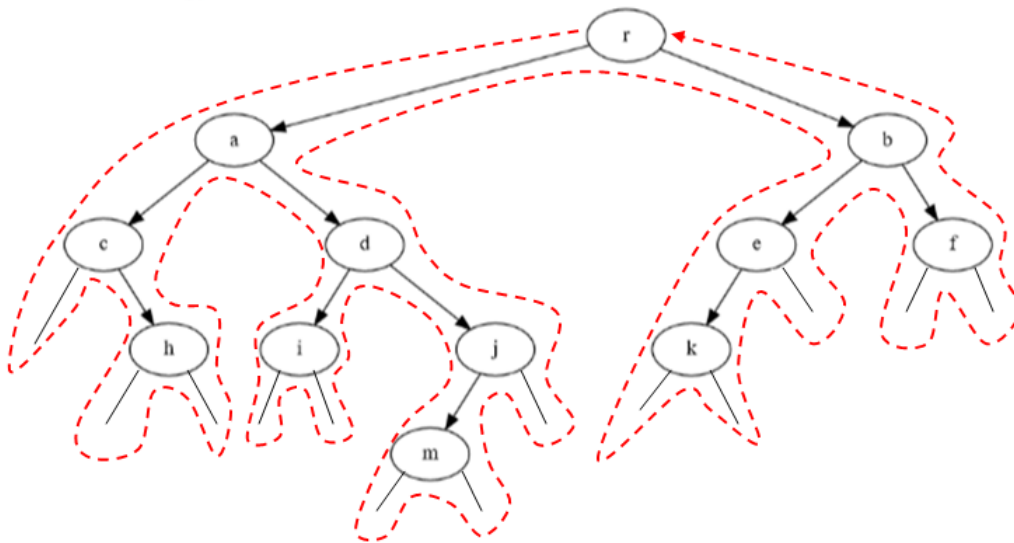
```
Fonction parcoursLargeur(arbre)
Données : arbre binaire
f une file vide
liste une liste vide
On met l'arbre dans la file
Tant que la file n'est pas vide faire
    On défile la file dans une variable tmp
    On ajoute tmp[0] à la liste
    Si tmp[1] n'est pas vide alors
        On enfile tmp[1]
    Si tmp[2] n'est pas vide alors
        On enfile tmp[2]
renvoyer liste
```

À FAIRE 3:

Implémenter cette fonction dans le code du "À faire 2". Vous aurez pris soin de récupérer le fichier `classeFile.py` ou `File.py` dans le cours sur les files pour pouvoir utiliser une file... Vous devez obtenir: ['r', 'a', 'b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'k', 'm']

Les parcours en profondeur

On se balade autour de l'arbre en suivant les pointillés



Dans le schéma ci-dessus, on a rajouté des "noeuds fantômes" pour montrer que l'on peut considérer que chaque noeud est visité 3 fois

- Une fois par la gauche.
- Une fois par en dessous.
- Une fois par la droite.

On peut définir les parcours comme suit :

DÉFINITION :

Dans un parcours **préfixe**, on liste le noeud la première fois qu'on le rencontre.
On le touche par la gauche en suivant la ligne

QUESTION 4:

Écrire les sommets dans un parcours préfixe.

.....

DÉFINITION :

Dans un parcours **infixe**, on liste le noeud la seconde fois qu'on le rencontre. (par le bas)
Ce qui correspond à: on liste chaque noeud ayant un fils gauche la seconde fois qu'on le voit et chaque noeud sans fils gauche la première fois qu'on le voit.

QUESTION 5:

Écrire les sommets dans un parcours infixé.

.....

DÉFINITION :

Dans un parcours **suffixe**, on note le noeud la dernière fois qu'on le voit. (par la droite)

QUESTION 6:

Écrire les sommets dans un parcours suffixé.

.....

QUESTION 7:

Voici 3 algorithmes récursifs, dire pour chacun d'entre eux à quel parcours il correspond.

```
Fonction parcours(arbre)
Données : arbre binaire
Si L'arbre n'est pas vide alors
    |   parcours(sous-arbre gauche)
    |   parcours(sous-arbre droit)
    |   Afficher : La racine de l'arbre
```

.....

.....

```
Fonction parcours(arbre)
Données : arbre binaire
Si L'arbre n'est pas vide alors
    |   Afficher : La racine de l'arbre
    |   parcours(sous-arbre gauche)
    |   parcours(sous-arbre droit)
```

.....

.....

```
Fonction parcours(arbre)
Données : arbre binaire
Si L'arbre n'est pas vide alors
    |   parcours(sous-arbre gauche)
    |   Afficher : La racine de l'arbre
    |   parcours(sous-arbre droit)
```

.....

.....

À FAIRE 4:

Implémenter ces trois fonctions dans le code du "À faire 3" et confirmer les réponses des questions 4, 5, 6, et 7 .

REMARQUE :

Tout ce que nous venons de faire peut être fait uniquement avec la variable **racine** qui est un dictionnaire représentant l'arbre de manière tout aussi efficace que la représentation par tableau de tableaux de arbre1.

Une troisième façon de faire...avec une classe

Nous allons créer une classe `Noeud` dont les attributs d'instances seront :

- Le nom (ou valeur) de la racine.
- Son fils gauche (vide par défaut).
- Son fils droit (vide par défaut).

De plus on rajoute une méthode spécifique, qui permet d'afficher la racine du noeud avec la fonction `print()`

Ci-dessous la classe `Noeud` et la représentation de l'arbre. (La racine possède deux fils qui sont eux mêmes des noeuds possédant deux fils qui.....)

```
class Noeud:

    # Le constructeur
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
    # Méthode qui permet d'afficher la valeur
    # de la racine avec la fonction print
    def __str__(self):
        return str(self.value)
```

```
k = Noeud('k')
f = Noeud('f')
e = Noeud('e', k, None)
b = Noeud('b', e, f)
m = Noeud('m')
j = Noeud('j', m, None)
i = Noeud('i')
d = Noeud('d', i, j)
h = Noeud('h')
c = Noeud('c', None, h)
a = Noeud('a', c, d)
arbre = Noeud('r', a, b)
```

```
print(arbre) # affiche r
```

REMARQUE :

On aurait pu écrire la représentation de l'arbre sur une seule ligne...mais c'est assez compliqué de ne pas se perdre...

```
arbre=Noeud('r',Noeud('a',Noeud('c',None,Noeud('h')),Noeud('d',Noeud('i'),
Noeud('j',Noeud('m'),None))),Noeud('b',Noeud('e',Noeud('k',None,None),None),
Noeud('f')))
```

À FAIRE 5:

Écrire une méthode `estFeuille` qui renvoie vrai si le noeud est une feuille et faux sinon.

La hauteur de l'arbre: Reprenons notre fonction `hauteur(arbre)` du "À faire 2"

```
def hauteur(arbre ) :  
    if arbre == [] :  
        return -1  
    else :  
        h1 = 1 + hauteur( arbre[1] )  
        h2 = 1 + hauteur( arbre[2] )  
        return max(h1, h2)
```

Adaptons là à notre contexte :

```
def hauteur2(arbre) :  
    if not arbre:  
        return -1  
    else :  
        h1 = 1 + hauteur2( arbre.left )  
        h2 = 1 + hauteur2( arbre.right )  
        return max(h1, h2)
```

À FAIRE 6:

Rajouter cette fonction au code précédent et vérifier que la hauteur de notre arbre y est toujours de 4.

Et si on en faisait une méthode...

Généralement pour transformer une fonction en méthode, il suffit de remplacer la variable (`arbre` ici) par `self`, cependant du fait de la propriété récursive de la fonction, il faudra en plus s'assurer que les arbres gauche et droit existent.
voici la méthode :

```
def hauteur(self):  
    h1=0  
    h2=0  
    if not self:  
        return -1  
    else:  
        if self.left:  
            h1 = 1 + self.left.hauteur()  
        if self.right:  
            h2 = 1 + self.right.hauteur()  
        return max(h1, h2)
```

À FAIRE 7:

Rajouter cette méthode au code précédent et tester là avec la commande :
`print(arbre.hauteur())`.

Les parcours:

Voici les trois fonctions de parcours :

```
def parcours_infixe(arbre):
    if arbre:
        parcours_infixe(arbre.left)
        print(arbre, end=' ')
        parcours_infixe(arbre.right)

def parcours_prefixe(arbre):
    if arbre:
        print(arbre, end=' ')
        parcours_prefixe(arbre.left)
        parcours_prefixe(arbre.right)

def parcours_suffixe(arbre):
    if arbre:
        parcours_suffixe(arbre.left)
        parcours_suffixe(arbre.right)
        print(arbre, end=' ')
```

À FAIRE 8:

Votre mission :

Transformer en méthodes ces trois fonctions.

REMARQUE :

Vaut-il mieux utiliser une méthode ou une fonction?

Cela dépend en fait de ce que l'on souhaite faire, dans notre cas, si c'est juste pour afficher les parcours, les méthodes suffisent. Mais si c'est pour faire quelque chose avec les noeuds, une fonction sera plus efficace car nous pourrons stocker ces noeuds dans une liste.

Comme le montre le code ci-dessous

```
liste_parcours_infixe=[]

def parcours_infixe(arbre):
    if arbre:
        parcours_infixe(arbre.left)
        liste_parcours_infixe.append(arbre.value)
        parcours_infixe(arbre.right)
    return liste_parcours_infixe
```