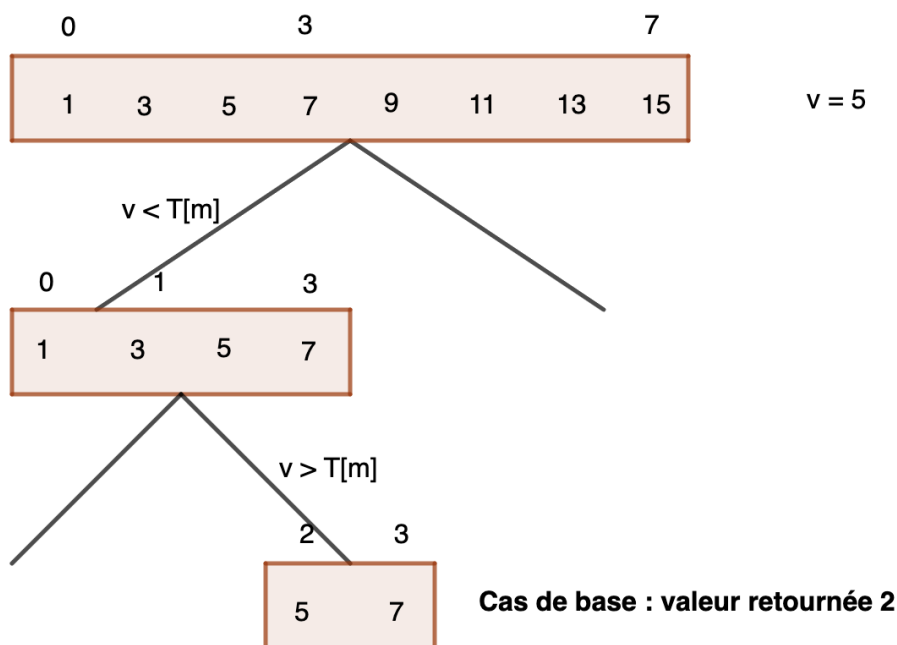


## Arbres

Nous avons vu dans le cours diviser pour régner que la **recherche dichotomique** était plus efficace que la **recherche séquentielle**.

La recherche séquentielle est associée à une structure linéaire alors que la recherche dichotomique est associée à une structure d'**arbre binaire**

par exemple si on recherche la valeur 5 dans la liste ci-dessous la succession des tests introduit un arbre binaire



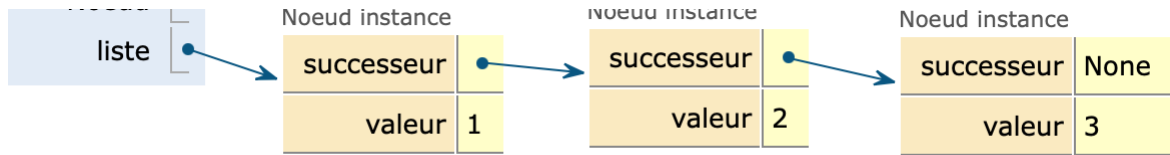
### 1 Arbres binaires

Avec Python Tutor on visualise une liste chaînée définie par

```
class Noeud:
```

```
    def __init__(self, v, s):
        self.valeur = v
        self.successeur = s
```

```
liste = Noeud(1, Noeud(2, Noeud(3, None)))
```



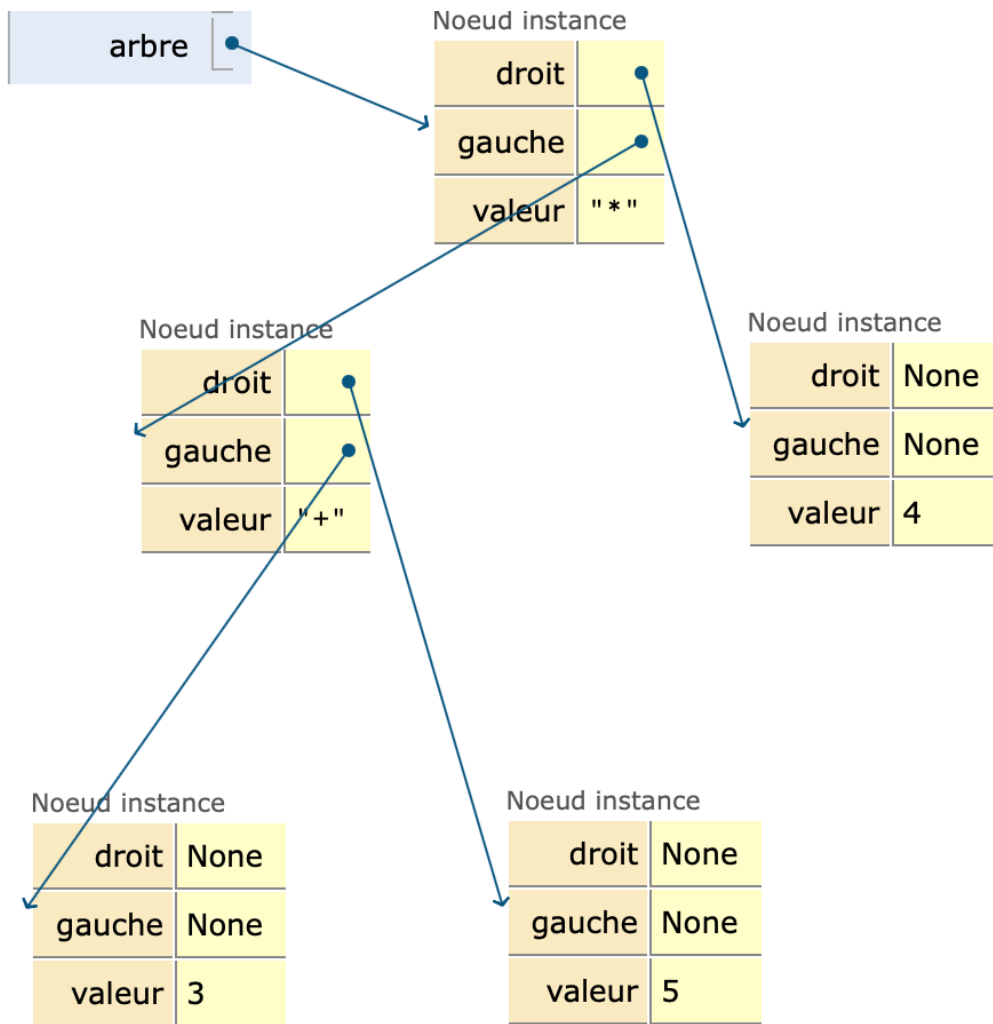
A la place d'un seul attribut `successeur` on met deux successeurs `droit` et `gauche`, plus précisément on définit ainsi un arbre binaire par exemple pour représenter l'expression arithmétique

$$(3 + 5) * 4$$

```
class Noeud:
```

```
    def __init__(self, v, g, d):
        self.valeur = v
        self.gauche = g
        self.droit = d
```

```
arbre = Noeud('*', Noeud('+', Noeud(3, None, None),
Noeud(5, None, None)), Noeud(4, None, None))
```



## Vocabulaire

1. Un arbre est vide s'il ne contient aucun noeud, autrement dit en Python

```
def est_vide(arbre):
    return arbre is None
```

2. La **racine** de l'arbre est le noeud sur lequel pointe la variable `arbre`

Dans l'exemple ci-dessus le noeud dont la valeur est '\*' est la racine de l'arbre

3. Le sous arbre gauche d'un noeud est référencé par l'attribut `gauche` de ce noeud (idem pour le sous arbre droit)

4. Une **feuille** est un noeud dont les sous-arbres droit et gauche sont vides

Dans l'exemple ci-dessus les feuilles sont les noeuds dont les valeurs sont 3, 5 et 4

5. La **taille** d'un arbre est le nombre de noeuds de l'arbre, dans l'exemple ci-dessus la **taille** vaut 5

La taille se définit naturellement de manière **récursive** ainsi :

```
def taille(arbre):  
    #le cas de base est une feuille  
    if arbre is None :  
        return 1  
    return 1 + taille(arbre.gauche) + \  
        taille(arbre.droit)
```

6. La racine d'un arbre binaire a au plus deux successeurs (on dit aussi enfants ou fils) qui eux même peuvent avoir au plus deux fils

A la **deuxième génération** ou au **deuxième niveau** on a donc au plus  $1 + 2 + 2^2 = 2^3 - 1$  noeuds

A la  $n$  ième génération on a donc au plus  $1 + 2 + 2^2 + \dots + 2^n = 2^{n+1} - 1$  noeuds

Lorsque le nombre maximal de noeuds est atteint on dit que l'arbre binaire est **complet**

7. Pour caractériser le fait qu'un arbre est plus ou moins complet on introduit la **hauteur** d'un arbre **récursivement** ainsi

(a) La hauteur d'un arbre vide est 0

(b) La hauteur d'un arbre non vide est la plus grande hauteur entre le sous arbre gauche et le sous arbre droit

```

plus 1
def hauteur(arbre):
    #le cas de base est une feuille
    if arbre is None :
        return 0
    return 1+ max(hauteur(arbre.gauche) ,
                 hauteur(arbre.droit))

```

Dans l'exemple ci-dessus la hauteur vaut 3

Si un arbre est "linéaire" dans ce cas sa taille  $t$  est égale à sa hauteur  $h$ .

Dans le cas général on a :

$$h \leq t$$

A l'opposé lorsqu'un arbre binaire est complet on a :

$$t = 2^{h-1}$$

Dans le cas général on a donc  $h \leq t \leq 2^h - 1$

## 2 Parcours d'arbres binaires

La structure linéaire d'une liste impose deux parcours naturels, soit on parcourt la liste du début à la fin, soit de la fin au début

Qu'en est il pour un arbre binaire? Les principaux parcours sont :

1. **Le parcours infixe** : Récursivement dans l'ordre :

- (a) On parcourt le sous-arbre gauche
- (b) On traite la racine (affichage par exemple)
- (c) On parcourt le sous-arbre droit

Ce qui donne pour notre exemple, l'arbre qui représente l'expression arithmétique  $(3 + 5) * 4$  :

$$3 + 5 * 4$$

d'où la fonction récursive suivante

---

**Algorithme 1** : Parcours infixe d'un arbre

---

parcours\_infixe (A)

début

**Données** : Un arbre A

**Résultat** : Rien

1   **si** *non* (A est vide) **alors**

2       parcours\_infixe (A.gauche)

3       afficher valeur(A)

4       parcours\_infixe (A.droit)

5   **fin**

**fin**

---

traduit en Python

```
def parcours_infixe(arbre):
```

```
    """
```

```
        si arbre est vide on ne fait rien
```

```
    """
```

```
    if arbre is not None:
```

```
        parcours_infixe(arbre.gauche)
```

```
        print(arbre.valeur)
```

```
        parcours_infixe(arbre.droit)
```

## Remarque

On aurait pu commencer par le sous-arbre droit.

Il existe une sorte de convention qui fait commencer par le sous arbre gauche

## 2. Le parcours préfixe

Récursivement dans l'ordre :

- (a) On traite la racine (affichage par exemple)
- (b) On parcourt le sous-arbre gauche
- (c) On parcourt le sous-arbre droit

puis le sous arbre droit

Ce qui donne pour notre exemple :

\* + 3 5 4

### 3. Le parcours postfixe

Récurivement dans l'ordre :

- (a) On parcourt le sous-arbre gauche
- (b) On parcourt le sous-arbre droit
- (c) On traite la racine (affichage par exemple)

puis le sous arbre droit

Ce qui donne pour notre exemple :

3 5 + 4 \*

(On retrouve la notation **postfixée** vue en TP pour l'utilisation d'une pile pour l'évaluation d'une expression arithmétique)

### 4. Le parcours en largeur, parcours en profondeur

En Première nous avons vu l'algorithme de Dijkstra comme exemple d'application de la **stratégie gloutonne**, nous allons revoir cette année cet algorithme mais cette fois ci comme un parcours en largeur dans un graphe

Qu'est ce qu'un parcours en largeur dans un arbre ?

On visite les noeuds **niveau par niveau** en commençant par la racine et de la gauche vers la droite

Dans notre exemple cela donne

- (a) On visite la racine '\*'

- (b) puis au premier niveau de la gauche vers la droite '+'  
puis 4
- (c) puis au deuxième niveau 3 puis 5

D'un point de vue algorithmique on se sert d'une **file** pour enregistrer les noeuds d'un même niveau les uns à la suite des autres tous ensemble

Lors du défilement ils seront traités dans le même ordre où ils ont été enregistrés

---

**Algorithme 2** : Parcours en largeur d'un arbre

---

```
parcours_largeur (A)
début
    Données : Un arbre A
    Résultat : Rien
1   f = file_vide()
2   enfiler(f,A)
3   tant que non (est-vide(f)) faire
4       | x = defiler(f)
5       | afficher(x.valeur)
6       | si non x.gauche est vide alors
7       | | enfiler(f,x.gauche)
8       | fin
9       | si non x.droit est vide alors
10      | | enfiler(f,x.droit)
11      | fin
12  fin
fin
```

---

Traduit en Python :



```

def parcours_largeur(arbre):
    """
        arbre est non vide
    """
    f = file_vide()
    enfiler(f, arbre)
    while not est_vide(f):
        x = defiler(f)
        print(x.valeur)
        if x.gauche is not None:
            enfiler(f, x.gauche)
        if x.droit is not None:
            enfiler(f, x.droit)

```

Que se passe-t-il si au lieu d'une file on utilise une pile?  
 Quel type de parcours a-t-on avec la fonction suivante?

```

def parcours_surprise(arbre):
    """
        arbre est non vide
    """
    p = pile_vide()
    empiler(p, arbre)
    while not est_vide(p):
        x = depiler(p)
        print(x.valeur)
        if x.droit is not None:
            empiler(p, x.droit)
        if x.gauche is not None:
            empiler(p, x.gauche)

```

A l'affichage on obtient :

\* + 3 5 4

On retrouve le parcours préfixe

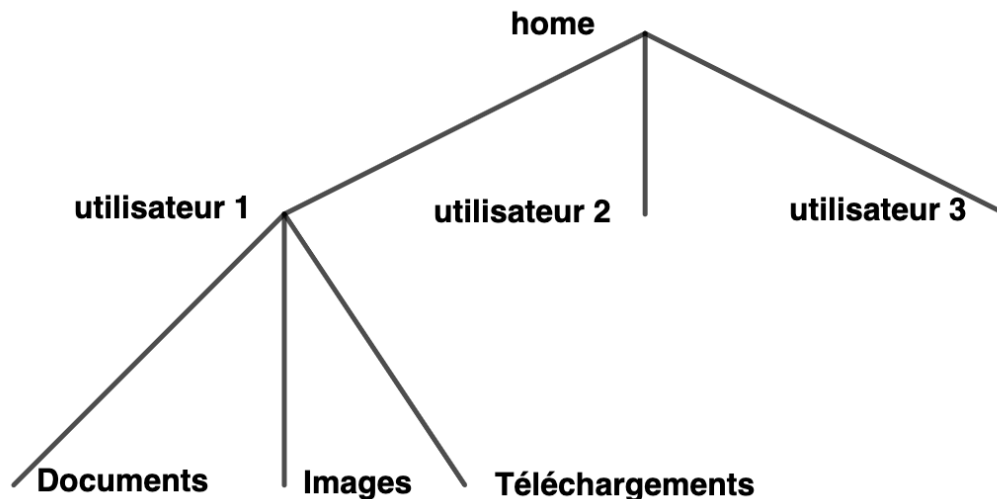
On parle aussi de **parcours en profondeur** car on essaie d'atteindre une feuille dès que c'est possible puis on remonte et on recommence

### 3 Arbres

On a déjà rencontré des arbres où un noeud peut avoir plus de deux successeurs et **il n'y a pas d'ordre entre les successeurs**

Par exemple :

1. L'arbre d'organisation des répertoires et des fichiers par un système d'exploitation, par exemple



2. Une page web est **structurée** par le langage HTML qui est un langage de balises

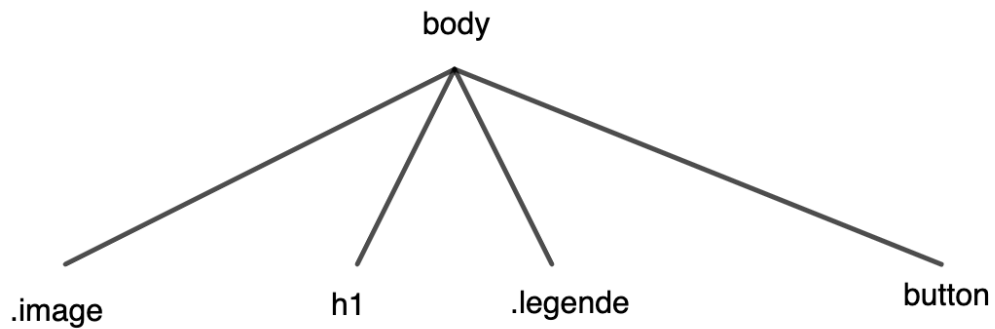
La plupart des balises sont ouvrantes et fermantes ce qui induit une relation hiérarchique entre les balises , que l'on peut matérialiser par un arbre appelé le DOM (document object model)

Voici la partie visible d'une page "simple" , entre les balises `<body>` et `</body>`

```
<body>
  

  <h1>Légende</h1>
  <p class='legende'>Image d'un chat</p>
  <button>Changer</button>
</body>
```

et voici le dom associé



## 4 Exercices

### Ex 1

1. Construire et dénombrer tous les arbres binaires de taille 1, 2, 3 et 4
2. On admet que le nombre d'arbres binaires de taille  $n$  vérifie :

$$a(n) = \frac{\binom{2n}{n}}{n+1}$$

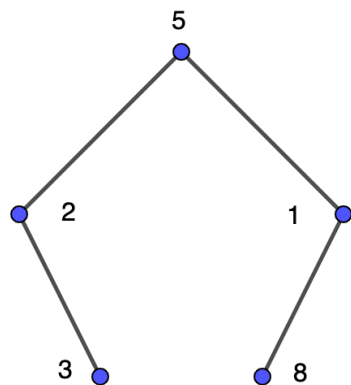
Ecrire une fonction Python qui renvoie le nombre d'arbres binaires de taille  $n$

### Ex 2

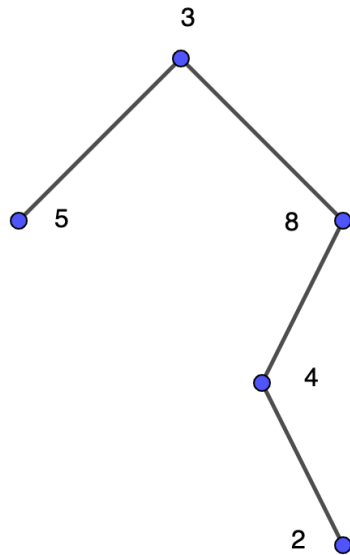
1. Dessiner l'arbre suite à l'exécution de

```
arbre = Noeud(1, Noeud(2, None, Noeud(3, None, None)),  
None)
```

Quelle est l'instruction correspondante à l'arbre représenté ci-dessous



3. Quelle est l'instruction correspondante à l'arbre représenté ci-dessous



### Ex 3

Implémenter la fonction `afficher(a)` permettant d'afficher un arbre binaire `a` sous la forme suivante :

(SAG Noeud SAD) où SAG est le sous arbre gauche du Noeud et SAD est le sous arbre droit

Par exemple l'arbre de l'expression arithmétique du cours sera affiché :

$$(((3)+(5))*(4))$$

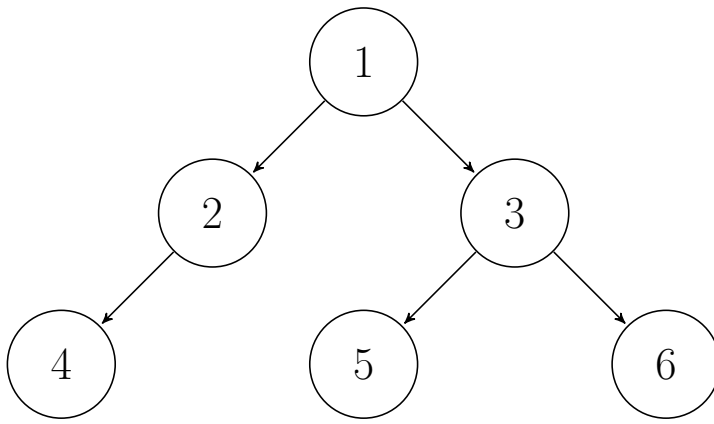
### Ex 4

Implémenter une fonction `parcours_postfixe(a)` qui affiche les éléments de l'arbre `a` dans un parcours postfixe

### Ex 5

Qu'observe-t-on à l'affichage pour un parcours de l'arbre suivant ?

1. préfixe
2. infixe
3. postfixe
4. en largeur



### Ex 6

En vous inspirant de ce qui a été fait avec les listes doublement chaînées écrire une classe `Noeud2` qui a un attribut supplémentaire `parent`

<b>BAC</b> Ex 4 Amérique du Nord 2021
---------------------------------------

<b>BAC</b> Ex 4 Métropole France 1 2022
---