# TP 10 – ARBRES

Info1.Algo1 - 2022-2023 Semestre Impair

## Rappels

Un **arbre binaire** est :

- soit un **arbre vide**.
- soit un **noeud** auquel on associe :
  - une **valeur**.
  - exactement 2 descendants (**gauche** et **droite**) qui sont eux-mêmes des arbres binaires.

L'**implémentation** utilisée est la suivante :

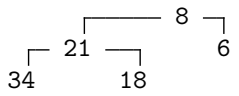
```
def creer_arbre_vide():
  return None

def creer_arbre(r,g,d):
  return r,g,d

def racine(arbre):
  return arbre[0]

def gauche(arbre):
  return arbre[1]

def droite(arbre):
  return arbre[2]

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

Ces 6 fonctions d'interface sont les seules autorisées pour manipuler les arbres. Elles sont présentes dans chaque fichier d'exercice.

**Exemple :**

`(8,(21,(34,None,None),(18,None,None)),(6,None,None))`

implémente l'arbre :




## Exercice 1 ★

Dans le fichier **ex01__proprietes.py** , compléter le corps des fonctions suivantes :

- La fonction récursive `calculer_taille` qui accepte en paramètre un arbre et retourne la **taille** (c'est à dire le nombre de noeuds) de cet arbre.
- La fonction récursive `calculer_hauteur` qui accepte en paramètre un arbre et retourne la **hauteur** (c'est à dire le nombre de niveaux) de cet arbre. **Convention pour cet exercice :** un arbre vide est de hauteur `0`.

## Exercice 2 ★

Dans le fichier **ex02__somme__maximum.py**, compléter le corps des fonctions suivantes :

- La fonction récursive `calculer_somme` qui accepte en paramètre un arbre dont les valeurs sont entières et retourne la somme des valeurs associées à tous ses noeuds.
- La fonction récursive `calculer_maximum` qui accepte en paramètre un arbre **non vide** dont les valeurs sont entières et retourne la plus grande des valeurs associées à ses noeuds.

## Exercice 3 ★

Dans le fichier **ex03__nb__occurrences.py**, compléter le corps de la fonction récursive `calculer_nb_occurences` qui accepte deux paramètres :

- `arbre` : un arbre.
- `valeur` : une valeur recherchée.

La fonction retourne le nombre de fois où la valeur recherchée apparaît dans l'arbre.

## Exercice 4 – Feuilles ★

**1)** Dans le fichier **ex04_feuilles.py**, compléter la fonction récursive `nombre_feuilles` qui accepte en paramètre un arbre et retourne le nombre de feuilles de cet arbre (un arbre vide n'a pas de feuilles).

**2)** Compléter la fonction récursive `feuille_droite` qui accepte en paramètre un arbre **non vide** et retourne la valeur associée à **la feuille la plus à droite** de cet arbre.

## Exercice 5 – Substitution ★

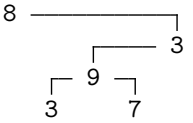
Dans le fichier **ex05_substituer.py**, compléter la fonction récursive `substituer` qui accepte en paramètres :

- un `arbre`.
- une valeur `ancienne` à remplacer.
- une valeur `nouvelle` par laquelle on doit la remplacer.

La fonction `substituer` construit un nouvel arbre dans lequel toute les occurrences de la valeur `ancienne` on été remplacées par la valeur `nouvelle`.

**Exemple :**

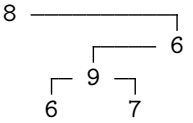
Si l'on considère l'arbre ci-dessous :




... et si :

`ancienne = 3` `nouvelle = 6`

... alors l'arbre retourné est :

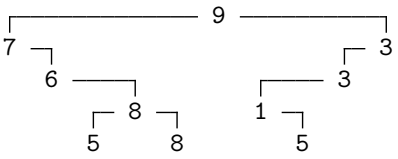



## Exercice 6 – Miroir ★

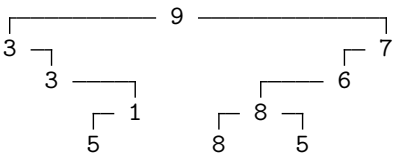
Dans le fichier **ex06_miroir.py**, compléter la fonction récursive `miroir` qui, étant donné un arbre, retourne l'arbre obtenu par symétrie verticale.

**Exemple :**

Si l'arbre est :




Son miroir est :




# Exercice 7 ★

> **Dans cet exercice on ne manipule pas des listes chaînées mais des listes natives Python (de type `list`) : l'usage de la concaténation + de deux listes est en particulier autorisé (voire recommandé).**

Dans le fichier **ex07_parcours_profondeur.py**, compléter le corps des fonctions suivantes :

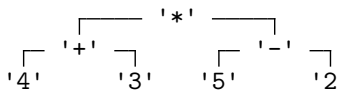
- La fonction récursive `parcours_profondeur_prefixe` qui accepte en paramètre un arbre et retourne la liste native Python (de type `list`) des valeurs obtenue lors d'un parcours en profondeur **préfixe** de cet arbre.
- La fonction récursive `parcours_profondeur_infixe` qui accepte en paramètre un arbre et retourne la liste (de type `list`) des valeurs obtenue lors d'un parcours en profondeur **infixe** de cet arbre.
- La fonction récursive `parcours_profondeur_suffixe` qui accepte en paramètre un arbre et retourne la liste (de type `list`) des valeurs obtenue lors d'un parcours en profondeur **suffixe** de cet arbre.

# Exercice 8 – Évaluation d'expressions algébriques ★★

On représente fréquemment les **expressions algébriques** par un arbre dont les noeuds sont :

- soit une **opération** (noeud ayant 2 sous-arbres)
- soit une **valeur** (qui sera donc une feuille de l'arbre).

Ainsi, l'expression `'(4+3)*(5-2)'` sera représentée par l'arbre suivant :




Dans cet exercice, les noeuds des arbres auront pour valeur associée une **chaîne de caractères** (de type `str`) qui pourra être :

- Soit l'une des 4 opérations binaires sur les entiers : `'+'`, `'-'`, `'*'`, ou `'//'`.
- Soit la représentation décimale d'un entier.

**Aucun des arbres testés ne contiendra d'autre forme de chaîne de caractères.**

Les opérations utilisées prenant toutes les quatre 2 opérandes, un arbre est considéré comme **valide** si :

- Chaque noeud ayant pour valeur associée une **opération** a **deux sous-arbres valides**.
- Chaque noeud ayant pour valeur associée la représentation décimale d'un **entier** est une **feuille**.

Un arbre **vide** n'est pas un arbre valide.

Dans le fichier **ex08_expression_algebrique.py**, compléter la fonction récursive `evaluer_expression` qui, étant donné un arbre donné en paramètre, évalue l'expression correspondante et retourne l'entier calculé. Si l'arbre n'est **pas valide** ou si une division par `0` a lieu, la fonction doit retourner `None`.

# Exercice 9 – Distance entre des noeuds dans un arbre ★★

**1)** On choisit de définir la distance entre deux mots (chaînes de caractères de type `str`) de la façon suivante :

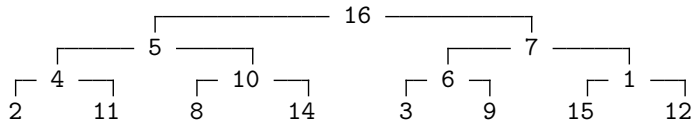
> **Définition :** On parcourt les deux chaînes de caractères simultanément tant que les caractères rencontrés sont égaux. Dès que le caractère rencontré est différent ou qu'on est arrivé à la fin d'un des deux mots, le **nombre de caractères restants** *(y compris les caractères différents rencontrés)* sera appelée **distance** entre les deux mots.

**Exemple :** La distance entre les deux mots `'avoir'` et `'avenue'`est égale à `7` : on parcourt les deux mots jusqu'à rencontrer deux caractères différentes (`'o'` et `'e'`) on compte alors le nombre d'éléments non-traités (`3` pour la première et `4` pour la seconde).

Dans le fichier **ex09_distance_dans_arbre.py**, compléter la fonction `distance_entre_mots` qui accepte en paramètres deux mots et retourne la distance ainsi calculée. **Cette fonction n'est pas nécessairement récursive.**

**2)** Dans cette question ainsi que dans la suivante, **on suppose que chaque noeud de l'arbre étudié a une valeur différente**. Une valeur est donc associée de manière univoque à un noeud et réciproquement. On peut alors décrire la position d'une valeur dans l'arbre par le chemin qui permet d'y accéder depuis la racine.

**Exemple :** Si l'on suppose que l'arbre est :




et que la valeur est `3`, le chemin à parcourir est, depuis la racine `16` :

- **droite** (`'d'`) pour arriver à `7`
- **gauche** (`'g'`) pour arriver à `6`
- **gauche** (`'g'`) pour arriver à la valeur `3`.

Ce chemin peut donc être décrit par la chaîne de caractères `'dgg'` (de type `str`).

Compléter la fonction récursive `chemin_vers_valeur` qui accepte en paramètres un `arbre` ainsi qu'une `valeur` et retourne la chaîne de caractères décrivant le chemin dans l'arbre qui mène à cette valeur.

- Si cette valeur est celle de la racine, la chaîne retournée est la chaîne vide `''`.
- Si la valeur n'est pas présente dans l'arbre, alors la fonction doit retourner `None`.

**3)** On définit la distance entre deux noeuds (et donc deux valeurs) d'un arbre de la façon suivante :

> **Définition :** La **distance entre deux noeuds** dans un arbre est le nombre de connexions qu'il faut parcourir pour aller de l'un à l'autre des noeuds.

**Exemple :** Dans l'arbre ci-dessus :

- la distance entre `5` et `4` est égale à 1.
- la distance entre `8` et `14` est égale à 2.
- la distance entre `11` et `1` est égale à 5.
- la distance entre un noeud et lui-même est toujours égale à 0.

Compléter la fonction `distance_dans_arbre` qui accepte en paramètre un `arbre` ainsi que deux valeurs `valeur1` et `valeur2` et retourne la distance entre les deux noeuds correspondants.

Si l'une des deux valeurs n'est pas présente dans l'arbre, la valeur retournée par la fonction `distance_dans_arbre` est `-1`.

**Indications :**

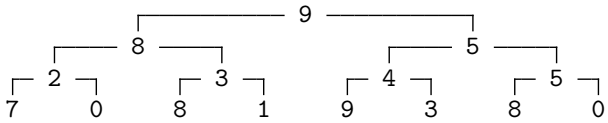
- Cette fonction n'a pas besoin d'être récursive ni d'utiliser de boucle.
- Il suffit de déterminer le chemin menant à chacune des deux valeurs puis d'utiliser la fonction `distance_entre_mots` sur ces deux chemins.

## Exercice 10 – Arbres parfaits ★★

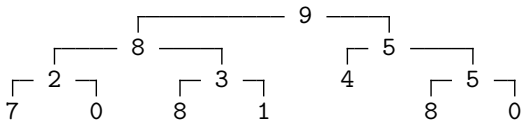
> **Définition :** Un **arbre binaire parfait** est un arbre binaire dans lequel toutes les **feuilles** sont **au même niveau** et où tous les autres noeuds ont **exactement deux descendants**.

**Exemples :**

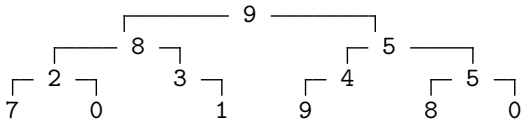
Arbre **parfait** :




Arbre **non parfait** *(toutes les feuilles ne sont pas au même niveau)*




Arbre **non parfait** *(certains noeuds n'ont qu'un seul descendant)*




Dans le fichier **ex10__arbres__parfaits.py**, compléter les fonctions récursives :

- `arbre_parfait_prefixe`
- `arbre_parfait_infixe`
- `arbre_parfait_suffixe`
- `arbre_parfait_largeur`

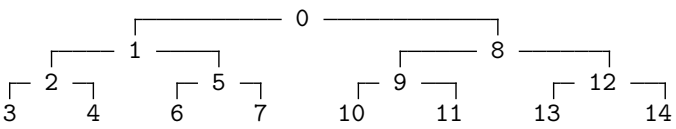
Ces 4 fonctions acceptent en paramètre l'entier `hauteur` qui détermine la hauteur de l'**arbre parfait** que la fonction devra retourner.

La valeur associée à chacun des noeuds des arbres retournés est la position (démarrant à `0`) de ce noeud dans chacun des parcours associés (en profondeur

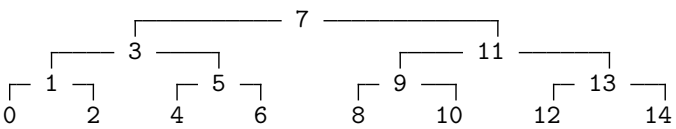
**préfixe**, **infixe**, **suffixe** et en **largeur**).

**Exemples :** Arbres retournés pour une hauteur égale à `4` :

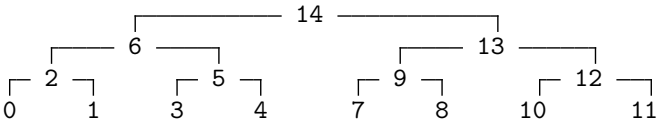
`arbre_parfait_prefixe` :



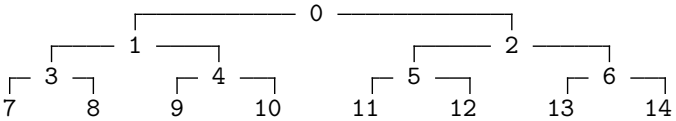

`arbre_parfait_infixe` :




`arbre_parfait_suffixe` :




`arbre_parfait_largeur` :




**Indication :**

On pourra utiliser le fait que la taille $t_n$ d'un arbre parfait de hauteur $n$ vérifie, pour tout entier $n \in \mathbb{N}$ :

$$t_n = 2^n - 1$$

**Pour aller plus loin :** sauriez-vous montrer cette formule en utilisant un raisonnement par récurrence ?