Lorsque nous écrivons des expressions arithmétiques nous utilisons la notation infixée (le **'in'** de **in**fixée indique que l'opérateur est **'entre'** les deux opérandes dans le cas d'un opérateur binaire). Ainsi pour écrire l'addition de 3 et 4, beaucoup parmi nous écrivent : `3 + 4`. L'opérateur + est bien écrit entre les deux opérandes 3 et 4.

La notation **post**fixée consiste à écrire l'opérateur **après** les deux opérandes. Ainsi en notation postfixée, l'addition de 3 et 4 s'écrit `3 4 +`. Cette notation postfixée est communément appelée notation polonaise inverse (la notation polonaise est **pré**fixée, la notation polonaise inverse est **post**fixée).

Pour toute la suite de ce sujet, les seuls opérateurs considérés sont +, -, * et /.

I : Se familiariser avec la notation postfixée d'expressions mathématiques

Pour évaluer une expression, par exemple celle-ci : `3 4 + 6 5 4 3 2 1 + - + - + -` il suffit de lire à partir de la gauche et – dès qu'on tombe sur un opérateur – de remplacer cet opérateur et les deux opérandes à sa gauche par le résultat du calcul.

Ici on commence par cet opérateur :

```
3 4 + 6 5 4 3 2 1 + - + - + -
    ↑
```

On remplace donc `3 4 +` ce qui donne :

```
    7 6 5 4 3 2 1 + - + - + -
```

On poursuit avec cet opérateur :

```
    7 6 5 4 3 2 1 + - + - + -
                  ↑
```

Ce qui donne :

```
    7 6 5 4 3     3 - + - + -
```

On poursuit avec cet opérateur :

```
    7 6 5 4 3     3 - + - + -
                    ↑
```

Ce qui donne :

```
    7 6 5 4         0 + - + -
```

en poursuivant ainsi on arrive finalement à ... `0`.

Remarque : L'avantage de cette notation postfixée est qu'elle ne nécessite aucun parenthésage.

Question 1 :
Évaluer les trois expressions mathématiques postfixes ci-dessous :

- `3 4 + 5 6 + *`
- `3 4 5 + + 6 *`
- `3 4 5 + * 6 +`

Question 2 :
Cette expression infixe : `(3 + 4) * 5 + 6 – (1 + 2)` peut se noter de plusieurs façons en notation postfixe.
Par exemple : `3 4 + 5 * 6 + 1 2 + -` ou encore `6 5 4 3 + * + 1 2 + -` .

Convertir les trois expressions mathématiques infixes ci-dessous en expressions mathématiques postfixes :

- `3 * (4 + 5) * 6 + 7`
- `3 + 4 * 5 – (6 + 7)`
- `(3 + 4) * (5 + 6 – 7)`

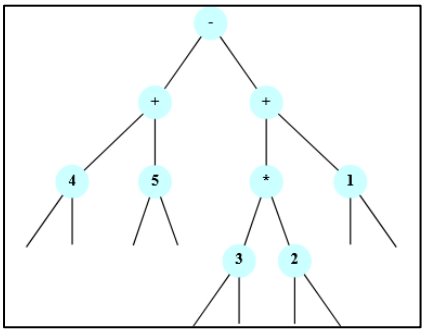
II : Expressions mathématiques représentées par des arbres binaires

Pour toute la suite de ce sujet, on décide d'implémenter en Python un arbre binaire à l'aide de la classe Noeud ci-contre.

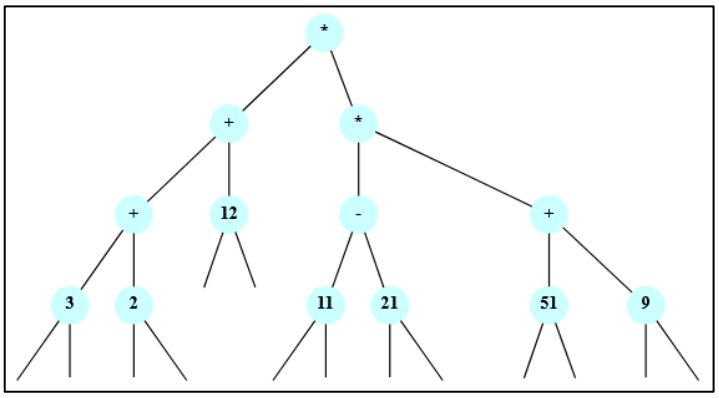
Un sous-arbre vide sera représenté par None.

```
class Noeud :
    def __init__(self, etiquette, gauche, droit):
        self.etiq = etiquette
        self.sag = gauche
        self.sad = droit

    def est_feuille(self):
        ...
```



Assez naturellement on peut associer des expressions mathématiques à des arbres binaires dans lesquels les feuilles sont des opérandes et les autres nœuds sont des opérateurs. Par exemple cet arbre binaire à droite correspond à l'expression mathématique postfixe `4 5 + 3 2 * 1 + -` , c'est-à-dire à l'expression mathématique infixe `(4 + 5) - ((3 * 2) + 1)`.



Question 3 :
Déterminer les expressions mathématiques postfixe et infixe associées à l'arbre binaire ci-contre à droite.

Question 4 :
Dessiner l'arbre `exp` défini par le code ci-contre.

```
f1 = Noeud(2, None, None)
f2 = Noeud(5, None, None)
f3 = Noeud(8, None, None)
f4 = Noeud(12, None, None)
f5 = Noeud(3, None, None)
n1 = Noeud("*", f1, f2)
n2 = Noeud("+", n1, f3)
n3 = Noeud("-", f4, f5)
exp = Noeud("+", n2, n3)
```

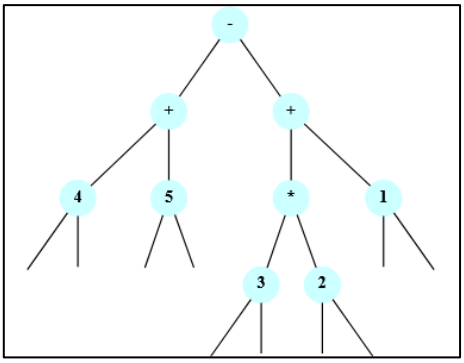
Question 5 :
Dans la classe Nœud quel est ou quels sont les attributs ?

Question 6 :
Recopier et compléter le code de la méthode `est_feuille` de la classe Noeud pour qu'elle renvoie `True` si le nœud est une feuille, et renvoie `False` sinon.

III : D'un arbre binaire vers les expressions mathématiques

On considère l'arbre binaire ci-contre. Si on parcourt cet arbre on obtient une succession de d'étiquettes des nœuds. Par exemple si on parcourt les noeuds de façon aléatoire on pourrait obtenir la succession d'étiquettes suivante :

`+ 2 5 + - 3 4 * 1`



Question 6 :

Donner, pour chacun des quatre différents parcours d'arbre ci-dessous, la succession d'étiquettes obtenues.

- parcours en largeur
- parcours en profondeur préfixe
- parcours en profondeur infixe
- parcours en profondeur postfixe (ou suffixe)

Question 7 :

Parmi les quatre parcours ci-dessus il y en a un qui permet d'obtenir une succession d'étiquettes correspondant à la notation **postfixe** de l'expression mathématique représentée par cet arbre binaire. De quel parcours s'agit-il ?

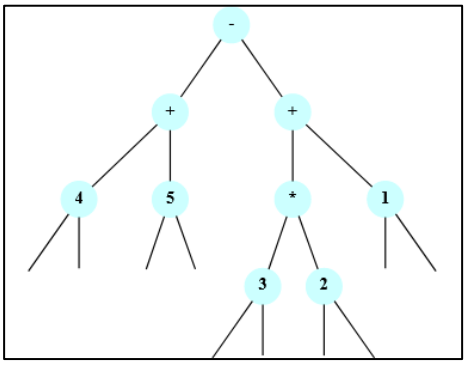
Pour toute la suite de ce sujet on utilise une structure de file d'ont l'interface est donnée ci-dessous :

Interface de la classe `File` :

```
File()     :  Crée une file vide.
enfile(el) :  Ajoute l'élément el à la queue de la file.
defile()   :  Supprime et renvoie l'élément en tête de la file.
              Déclenche une erreur si la file est vide.
est_vide() :  Renvoie True si la file est vide, False sinon.
```

L'expression mathématique représentée par l'arbre binaire ci-contre en notation infixe parenthésée est :

`(4 + 5) - ((3 * 2) + 1)`



Question 8 :

La fonction récursive incomplète ci-contre prend pour paramètres un arbre représentant une expression mathématique et une file initialement vide. Elle renvoie la file contenant les étiquettes de l'arbre et les parenthèses correspondant à la notation **infixe** parenthésée.

```
def infixe(arbre, file):
    if arbre != None:
        ...(1)
        parcours_g = infixe(arbre.sag, file)
        ...(2)
        file.enfile(arbre.etiq)
        ...(3)
        parcours_d = infixe(arbre.sad, file)
        ...(4)
    return file
```

Quels sont les deux emplacements parmi `(1)`, `(2)`, `(3)` et `(4)` où l'on doit placer les deux instructions A et B pour que la fonction soit correctement complétée ?

Instruction A :

```
if not arbre.est_feuille():
    file.append("(")
```

Instruction B :

```
if not arbre.est_feuille():
    file.append(")")
```

IV : D'une expression mathématique postfixe vers un arbre binaire

Pour toute la suite de ce sujet on utilise une structure de pile dont l'interface est donnée ci-dessous :

Interface de la classe `Pile` :

```
Pile()     :  Crée une pile vide.
empile(el) :  Empile l'élément el au sommet de la pile.
depile()   :  Supprime et renvoie l'élément au sommet de la pile.
              Déclenche une erreur si la pile est vide.
est_vide() :  Renvoie True si la file est vide, False sinon.
```

Question 9 :

Des deux acronymes LIFO et FIFO, indiquer lequel permet de décrire la structure de pile et lequel permet de décrire la structure de file. Donner la signification des quatre lettres qui composent ces acronymes.

On considère l'algorithme ci-dessous, écrit en pseudo-code, qui prend en argument une file de caractères correspondant à la notation postfixe d'une expression mathématique et qui renvoie l'arbre binaire associé.

Algorithme créer_arbre(*file_postfixe*)

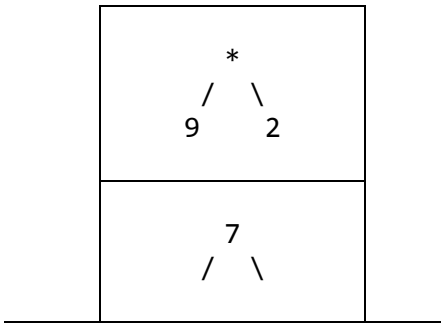
```
P ← créer une pile vide

Tant que file_postfixe n'est pas vide :
    élément ← défiler file_post_fixe
    Si élément est un entier :
        a ← construire l'arbre de racine élément n'ayant aucun sous-arbre
    Sinon :
        a_droite ← dépiler P
        a_gauche ← dépiler P
        a ← construire l'arbre de racine élément ayant pour sous-arbre gauche
            a_gauche et pour sous-arbre droit a_droite
    Empiler a dans P

a ← dépiler P
renvoyer a
```

Question 10 :

On applique cet algorithme sur la file contenant dans l'ordre la notation postfixe :
`"7", "9", "2", "*", "+", "94", "1", "-", "+"`
où "7" est en tête de liste.
À la fin de l'exécution du quatrième tour de la boucle «Tant que», la pile P contient deux éléments et son état est :



Indiquer le nombre d'éléments(s) contenu(s) dans la pile P et dessiner l'état de la pile P à la fin de l'exécution du cinquième tour de la boucle «Tant que».

Question 11 :

Dessiner l'arbre binaire obtenu en appliquant cet algorithme à la file contenant dans l'ordre la notation postfixe :
`"7", "9", "2", "*", "+", "94", "1", "-", "+"`
où 7 est en tête de liste.

V : Évaluation d'une expression mathématique donnée sous forme d'arbre

On souhaite écrire une fonction en langage Python qui permet d'évaluer une expression mathématique représentée par un arbre binaire implémenté avec la classe Noeud de la partie II.

On rappelle que les seuls opérateurs considérés sont +, -, * et /.

```
class Noeud :
    def __init__(self, etiquette, gauche, droit):
        self.etiq = etiquette
        self.sag = gauche
        self.sad = droit

    def est_feuille(self):
        '''Renvoie True si l'instance est une
        feuille et renvoie False sinon.'''
        ...
```

Question 12 :
Dans les arbres binaires représentant des expressions mathématiques, il y a deux catégories de nœuds :

- ceux qui contiennent des opérandes (des nombres),
- ceux qui contiennent des opérateurs (des opérations).

Quelle est la catégorie qui correspond aux feuilles des arbres binaires ?

Question 13 :
La fonction récursive `evaluer` donnée ci-dessous est écrite en Python. Cette fonction prend en paramètre un arbre binaire `arbre` représentant une expression mathématique et renvoie la valeur de cette expression mathématique.

```
def evaluer(arbre):
    if arbre.est_feuille():
        res = (instruction A)
    elif (instruction A) == "+":
        res = (instruction B) + (instruction C)
    elif (instruction A) == "-":
        res = (instruction B) - (instruction C)
    elif (instruction A) == "*":
        res = (instruction B) * (instruction C)
    elif (instruction A) == "/":
        res = (instruction B) / (instruction C)
    return res
```

Trois instructions seulement permettent de compléter ce code : instruction A, instruction B et instruction C.
Écrire sur la copie le code de chacune de ces trois instructions.