

## TP 15 - Les Arbres Enoncé-Corrigé

<b>TD n°15 : Structures de données - Les Arbres</b>	Thème 1 : Structures de donnée	Thème 4 : Algorithmique
<b>COURS et EXERCICES</b>		



### Le Programme en N.S.I en Terminale

- Définition d'un arbre binaire
- Mesure des arbres binaires,
- Représentation en Python,
- d'appréhender la complexité de la manipulation des listes,
- Algorithme des binaires.

Contenus	Capacités attendues	Commentaires
Arbres : structures hiérarchiques.	Identifier des situations nécessitant une structure de données arborescente..	On fait le lien avec la rubrique « algorithmique ».
Arbres binaires : nœuds, racines, feuilles, sous-arbres gauches, sous-arbres droits.	Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).	

La notion de liste chaînée explorée dans les TP précédents est parfaite pour structurer un ensemble d'éléments destiné à être énuméré séquentiellement.

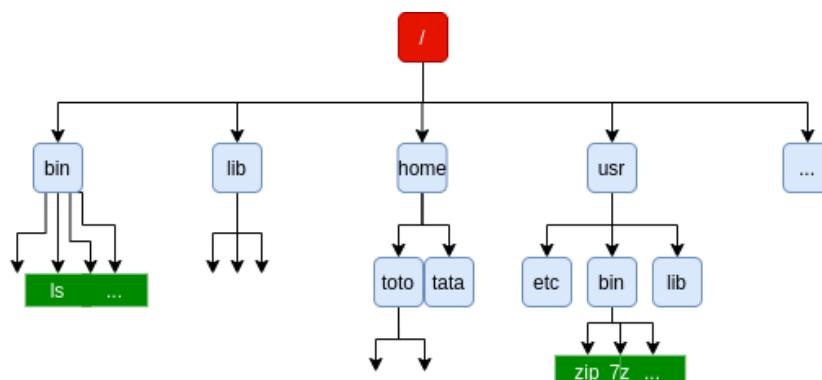
Comme on l'a vu avec la réalisation de piles et de files, cette structure permet également un accès simple au début de la séquence et éventuellement à certaines autres positions choisies.

Elle n'est pas en revanche pas adaptée aux accès ponctuels à des positions arbitraires dans la séquence, puisqu'il faut pour cela à chaque nouvel accès parcourir tous les maillons depuis la tête de liste jusqu'à la position cherchée, ce qui prend en moyenne un temps proportionnel au nombre d'éléments stockés dans la structure.

## I. Structures arborescentes

Les **structures arborescentes** forment une autre famille de structures chainées, dans lesquelles le nombre de sauts à effectuer pour aller depuis le point de départ jusqu'à une position souhaitée est potentiellement bien moindre.

Ces structures sont omniprésentes en informatique, par exemple l'arborescence des fichiers d'un ordinateur. Cette représentation des fichiers permet notamment, partant d'un répertoire racine et voyageant de répertoire en sous-répertoire, d'accéder en un petit nombre d'étapes à n'importe quel fichier choisi parmi des dizaines de milliers, pour peu qu'on aille dans la bonne direction.



Ce principe d'un point de départ unique à partir duquel une structure chainée se scinde à chaque étape en plusieurs branches donne une idée générale de la structure d'arbre en informatique, qui est à la base d'innombrables structures de données. Cette structure permet en outre une organisation hiérarchique de l'information, ce qui la rend utile pour représenter des programmes, des formules logique, le contenu de pages web,etc...

Dans ce chapitre, nous allons nous concentrer sur les arbres binaires , une forme particulière mais très répandue de structure arborescente.

## II. Arbres quelconques

### Vocabulaire

Dans la terminologie informatique, on utilise les termes de

- **feuille** pour les informations élémentaires,
- **noeud** pour chaque embranchement de l'arbre,
- **racine** pour le(s) noeud(s) principal(aux).

**Attention :** l'analogie avec les arbres réels peut s'avérer trompeuse. Les arbres - en informatique - sont le plus souvent représentés avec la racine en haut, puis les noeuds, et les feuilles en bas.

Il s'agit d'une structure de données abstraite permettant de représenter une collection de données par des noeuds organisés de manière hiérarchique : il y a un (parfois plusieurs) noeud racine et chaque noeud dépend d'un

antécédent (sauf la racine) et a des descendants (sauf les feuilles).

### Vocabulaire

Dans le vocabulaire des arbres on utilise les termes *père* et *fils* pour désigner respectivement un antécédent et les descendants.

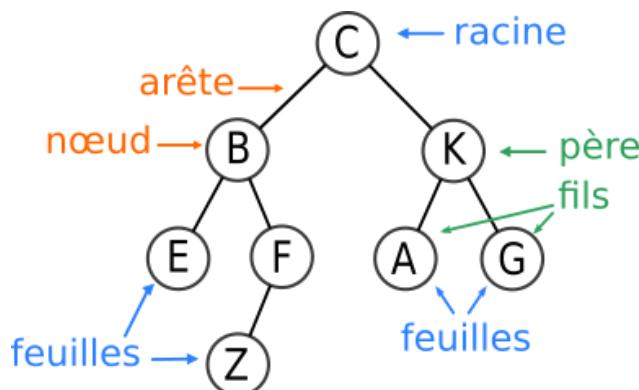
- **Père** : chaque noeud possède exactement un seul noeud *père*, celui dont il est issu, à l'exception de la racine qui n'en a pas.
- **Fils** : chaque noeud peut avoir un nombre arbitraire de noeuds *fils*, dont il est le père.

Ainsi, avec ces définitions :

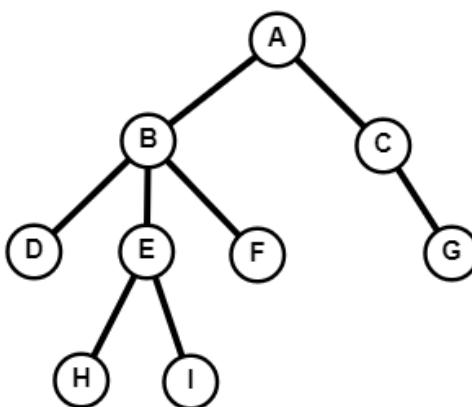
- les **feuilles** sont les noeuds qui n'ont pas de fils,
- un noeud qui n'a pas de père s'appelle une **racine**.

Tous les noeuds qui ne sont pas des feuilles sont appelés des **noeuds internes** (et les feuilles parfois appelées des **noeuds externes**).

L'intérêt des arbres est d'y stocker de l'information. Pour cela, chaque noeud peut contenir une ou plusieurs valeurs. L'information portée par un noeud s'appelle l'**étiquette** du noeud (ou la *valeur*, ou la *clé*).



Exemple :



Dans cet arbre :

- La racine est le noeud A.

- Le noeud B possède 3 fils (les noeuds D, E et F), le noeud C possède un fils (le noeud G), le noeud F ne possède aucun fils.
- Le noeud B a pour père le noeud A.
- Les feuilles sont les noeuds D, H, I, F et G (ceux qui n'ont pas de fils).

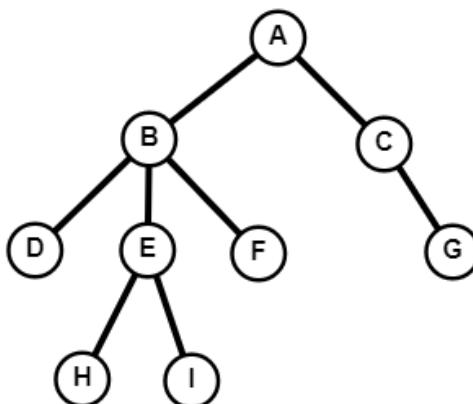
## Caractéristiques d'un arbre

### Définition

- La **taille** d'un arbre est le nombre de noeuds qu'il possède.
- La **profondeur** d'un noeud est la longueur du chemin le plus court entre ce noeud et la racine (la racine a donc une profondeur égale à 0).
- La **hauteur** d'un arbre est la profondeur maximale de ses noeuds (elle vaut 0 pour l'arbre réduit à sa racine et  $-1$  par convention pour un arbre vide).

**Attention :** On trouve aussi dans la littérature, que la profondeur de la racine est égale à 1, ce qui modifie la hauteur de l'arbre également puisqu'alors l'arbre réduit à la racine a pour hauteur 1 et l'arbre vide a pour hauteur 0. Les deux définitions se valent, il faut donc bien lire celle qui est donnée.

### Exemple :



- La **taille** de l'arbre est égale à 9 (il possède 9 noeuds : 4 noeuds internes et 5 feuilles).
- Le noeud E a une **profondeur** égale à 2 (le chemin A-B-E a une longueur égale à 2).
- La **hauteur** de l'arbre est égale à 3 (la profondeur maximale est égale à 3, c'est celle des noeuds les plus profonds : H et I).

*Dans la suite, on ne s'intéressera qu'aux arbres dont les noeuds ont au plus deux fils.*

## II. Définition et propriétés des arbres binaires

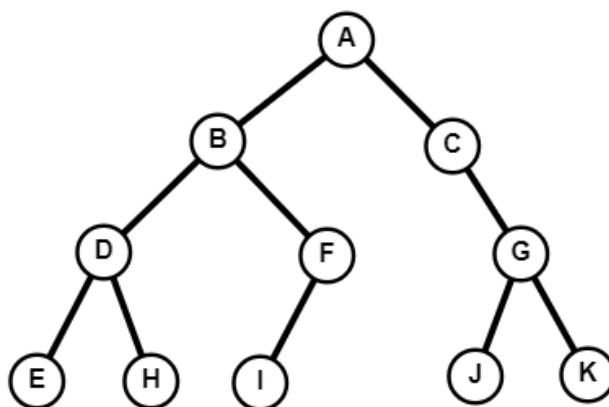
Seuls les **arbres binaires** sont au programme de Terminale NSI.

### Définition et vocabulaire spécifique

#### Arbre binaire

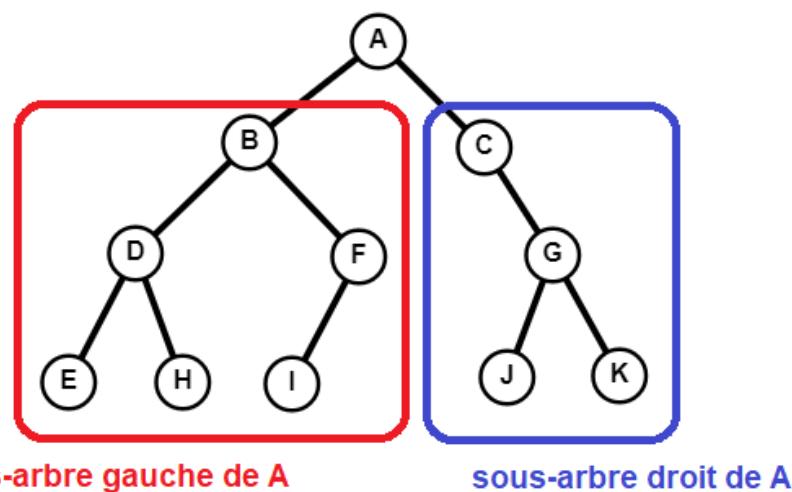
Un **arbre binaire** est un arbre dont tous les noeuds ont au plus deux fils.

L'arbre vu dans le paragraphe précédent n'est pas binaire car le noeud B possède 3 fils. En revanche, l'arbre suivant est binaire.



Les définitions vues précédemment pour des arbres quelconques restent bien évidemment valables pour les arbres binaires. Dans le cas d'un arbre binaire, chaque noeud possède deux sous-arbres, éventuellement vides, que l'on appelle **sous-arbre gauche** et **sous-arbre droit**.

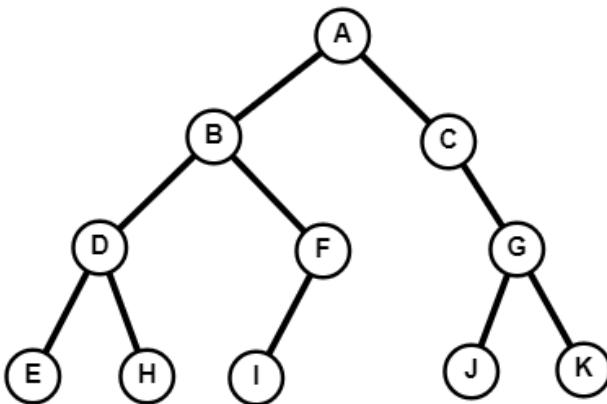
Par exemple, dans le cas de l'arbre binaire précédent, le noeud A possède un sous-arbre gauche et un sous-arbre droit comme la figure suivante le montre.



Les sous-arbres gauche et droit de A sont eux-mêmes des arbres dont les racines sont respectivement B et C. Ces noeuds possèdent eux-mêmes des sous-arbres gauche et droit. Par exemple, le noeud C possède un sous-arbre gauche, qui est vide, et un sous-arbre droit qui est l'arbre dont la racine est G. Ainsi de suite...

### Exercice n°1 : Vocabulaire

Voici un arbre :



1. Quelle est la taille de cet arbre ?
2. Listez les noeuds internes puis les feuilles.
3. Quelle est la hauteur de cet arbre ?
4. Quels sont les fils du noeud B ?
5. Quel est le sous-abre gauche du noeud B ?
6. Quel est le sous-abre gauche du noeud C ?

Exercice n°2 :

On considère l'expression suivante :  $A = (2 + 4) \times 2 - 32 + 1$

Représenter cette expression par un arbre binaire dans lequel les noeuds sont les opérations et les feuilles, les nombres. Cela présente beaucoup d'avantages pour calculer l'expression car une fois l'expression écrite sous forme d'arbre, l'algorithme permettant de l'évaluer est aisé.

Exercice n°3 :

$((3 + 12) \times 5 + (4 + 17) \times 6) / (7 + (8 \times 4))$

Représenter cette expression par un arbre binaire dans lequel les noeuds sont les opérations et les feuilles, les nombres.

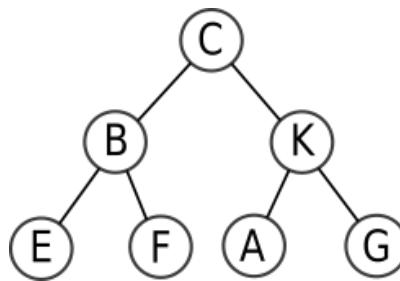
Exercice n°4 :

$(2 * (x + y)) / (x - 3 * y) + 2 * x$

Représenter cette expression par un arbre binaire dans lequel les noeuds sont les opérations et les feuilles, les nombres.

### Cas des arbres binaires complets

On rencontre très souvent des arbres binaires dits **complets** parce qu'aucun des fils gauche ou droit n'est manquant.



**Taille d'un arbre complet de hauteur  $h$  :**

$$1 + 2 + 2^2 + 2^3 + \cdots + 2^{h-1} = 2^h - 1$$

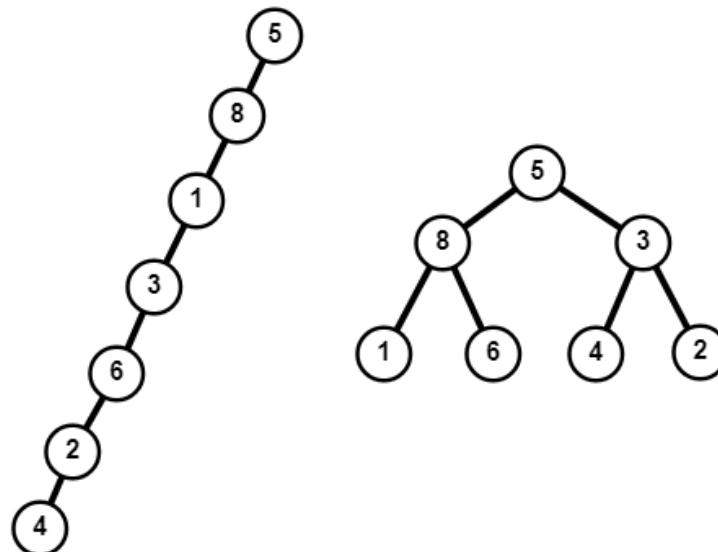
**preuve à connaître :**

ceci est la somme  $S$  des  $h$  premiers termes d'une suite géométrique de raison 2 et de premier terme 1, d'où  $S = \frac{1 - 2^h}{1 - 2} = 2^h - 1$ .

Un arbre complet de hauteur  $h$  (en prenant la convention que l'arbre vide a pour hauteur 0) a donc une taille égale à  $2^h - 1$ .

### Encadrement de la taille d'un arbre binaire

La hauteur d'un arbre binaire est la profondeur maximale de ses noeuds. Cependant un arbre binaire d'une taille donnée peut avoir un aspect totalement différent. En effet, les deux arbres binaires suivants sont de même taille (égale à 7) mais ont des "formes" très différentes.



**Remarque :** On en déduit une inégalité classique sur l'encadrement de la taille  $t$  d'un arbre binaire (non nécessairement complet) de hauteur  $h$  :

$$h \leq t \leq 2^h - 1$$

### III. Implémentations d'un arbre binaire

#### Activité 1 : Implémentation avec des listes Python

On peut construire un arbre binaire non vide comme un noeud composé de deux sous-arbres. Pour annoter la structure de l'arbre avec des informations, on utilise des étiquettes pouvant être enregistrées à chaque noeud. On peut ensuite parcourir un arbre par l'accès à son étiquette et à ses sous-arbres droit et gauche. Un prédictat permet de distinguer les feuilles des noeuds.

##### Interface

On peut ainsi spécifier un arbre binaire par le type abstrait suivant :

- Constructeur : `noeud : Etiquette x Arbre binaire x Arbre binaire -> Arbre binaire`
- Sélecteurs :
  - `droit : Arbre binaire -> Arbre binaire`
  - `gauche : Arbre binaire -> Arbre binaire`
  - `etiquette : Arbre binaire -> Etiquette`
- Prédicat : `est_feuille : Arbre binaire -> Booléen`

On choisit ici de représenter un arbre binaire par une liste de trois éléments `[etiquette, arbre_gauche, arbre_droit]` où `arbre_gauche` et `arbre_droit` désignent les sous-arbres gauche et droit du noeud `etiquette`. L'arbre vide est représenté par une liste vide.

##### Question 1 :

Dessinez l'arbre représenté par la liste suivante.

```
[4, [2, [5, [], []], [1, [], []]], [3, [], [6, [], []]]]
```

##### Question 2 :

###### Enoncé

Complétez les 5 fonctions suivantes qui implémentent le type abstrait `Arbre binaire` avec cette représentation par des listes Python.

```
# à vous de jouer !

def noeud(etiquette, arbre_gauche, arbre_droit):
    """Crée et renvoie l'arbre binaire"""
    pass

def etiquette(arbre):
    """Renvoie l'étiquette de l'arbre binaire arbre"""
    pass
```

```

def gauche(arbre):
    """Renvoie le sous-arbre gauche de l'arbre binaire arbre"""
    pass

def droit(arbre):
    """Renvoie le sous-arbre droit de l'arbre binaire arbre"""
    pass

def est_feuille(arbre):
    """Renvoie True si et seulement si l'arbre binaire arbre est une feuille"""
    pass

```

solution

```

# à vous de jouer !

def noeud(etiquette, arbre_gauche, arbre_droit):
    """Crée et renvoie l'arbre binaire"""
    return [etiquette,noeud_gauche,noeud_droit]

def etiquette(arbre):
    """Renvoie l'étiquette de l'arbre binaire arbre"""
    return arbre[0]

def gauche(arbre):
    """Renvoie le sous-arbre gauche de l'arbre binaire arbre"""
    return arbre[1]

def droit(arbre):
    """Renvoie le sous-arbre droit de l'arbre binaire arbre"""
    return arbre[2]

def est_feuille(arbre):
    """Renvoie True si et seulement si l'arbre binaire arbre est une feuille"""
    if gauche(arbre)==[] and droit(arbre)==[]:
        return True
    else:
        return False

```

### Question 3 :

1. Créez l'arbre, noté `a1`, de la question 1 en utilisant la fonction `noeud`. Vérifiez que la représentation est bien celle donnée dans la question 1.
2. Ecrivez les instructions permettant :
  - d'afficher le sous-arbre gauche de `a1`,
  - d'afficher l'étiquette de la racine du sous-arbre droit de `a1`,
  - d'accéder au sous-arbre droit du sous-arbre gauche de `a1`.

```
# à vous de jouer !
# question 3.1

# question 3.2
```

**Remarque:** Si on suppose que l'arbre n'est pas modifiable en place, on pourrait écrire la même implémentation avec des tuples plutôt que des listes Python.

## Activité 2: En utilisant la Programmation Orientée Objet

Le but est d'obtenir l'interface ci-dessous.

```
a = Arbre(4) # pour créer l'arbre dont le nœud a pour valeur 4,
              # et dont les sous-arbres gauche et droit sont None

a.ajout_gauche(3) # pour donner la valeur 3 au nœud du sous-arbre gauche de a
a.ajout_droit(1) # pour donner la valeur 1 au nœud du sous-arbre droit de a

a.affiche_droit() # pour accéder au sous-arbre droit de a
a.affiche_gauche() # pour accéder au sous-arbre gauche de a

a.valeur() # pour accéder à la valeur du nœud de l'arbre a
```

Il est à remarquer que ce que nous allons appeler «Arbre» est en fait un nœud et ses deux fils gauche et droit.

### Question 1 :

Dessinez l'arbre créé par les instructions suivantes :

```
a = Arbre(4)
a.ajout_gauche(3)
a.ajout_droit(1)
a.droit.ajout_gauche(2)
a.droit.ajout_droit(7)
a.gauche.ajout_gauche(6)
a.droit.droit.ajout_gauche(9)
```

```
class Arbre:
    def __init__(self,valeur):
        """Initialisation de l'arbre racine + sous-arbre gauche et sous-arbre droit"""
        self.v=valeur
        self.gauche=None
        self.droit=None

    def ajout_gauche(self,val):
        """On ajoute valeur dans le sous-arbre gauche sous la forme [val,None,None]"""
        self.gauche=Arbre(val)

    def ajout_droit(self,val):
        """ On ajoute valeur dans le sous-arbre droit sous la forme [val,None,None]"""
        self.droit=Arbre(val)

    def affiche(self):
        """permet d'afficher un arbre"""
        if self==None:
            return None
        else :
            return [self.v,Arbre.affiche(self.gauche),Arbre.affiche(self.droit)]
```

```

def taille(self):
    if self==None:
        return 0
    else :
        return 1+Arbre.taille(self.gauche)+Arbre.taille(self.droit)

def hauteur(self):
    if self==None:
        return 0
    elif self.gauche==None and self.droit==None:
        return 0
    else :
        return 1+max(Arbre.hauteur(self.gauche),Arbre.hauteur(self.droit))

def get_gauche(self):
    return self.gauche

def get_droit(self):
    return self.droit

def get_valeur(self):
    if self==None:
        return None
    else:
        return self.v

```

```

a = Arbre(4)
a.ajout_gauche(3)
a.ajout_droit(1)
a.droit.ajout_gauche(2)
a.droit.ajout_droit(7)
a.gauche.ajout_gauche(6)
a.droit.droit.ajout_gauche(9)
print(a.affiche())
a.get_droit().affiche()

```

[4, [3, [6, None, None], None], [1, [2, None, None], [7, [9, None, None], None]]]

[1, [2, None, None], [7, [9, None, None], None]]

Pour vérification :

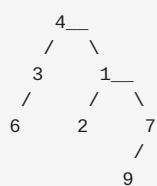
```

import binarytree as bt
from binarytree import Node

root = Node(4)
root.left = Node(3)
root.right = Node(1)
root.right.left = Node(2)
root.right.right = Node(7)
root.left.left = Node(6)
root.right.right.left = Node(9)

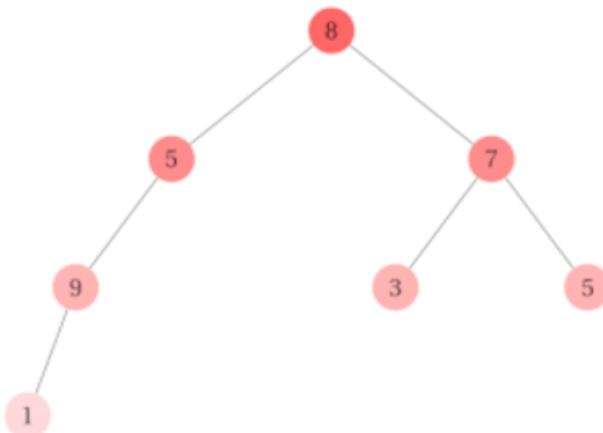
print(root)

```



**Question 2 :**

Implémenter cet arbre avec notre implémentation puis avec la bibliothèque disponible.



## IV. Parcours d'un arbre binaire

Un arbre binaire (abrégé AB dans la suite) est un arbre dont les noeuds possèdent au plus deux fils. Ainsi, un arbre binaire non vide peut être défini comme un noeud, appelé *racine* possédant un *sous-arbre gauche* et un *sous-arbre droit* (éventuellement vides) qui sont eux-mêmes des arbres binaires.

Cette définition étant récursive, la plupart des traitements sur les AB sont naturellement récursifs : on traite un noeud courant et on demande à traiter les noeuds fils. On a déjà écrit de tels algorithmes pour calculer la taille ou la hauteur d'un AB.

On ne s'était alors pas soucié de l'ordre dans lequel tous les noeuds étaient visités.

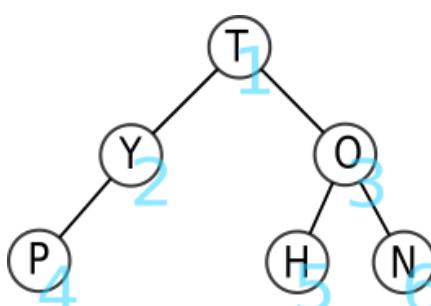
Dans cette partie, nous allons notamment voir des algorithmes permettant d'explorer, récursivement ou non, tous les noeuds d'un AB mais dans un ordre prédéfini. Nous verrons dans un second temps une structure de données appelée **arbre binaire de recherche**, qui est un AB particulier permettant de stocker des éléments de façon à rendre leur recherche très efficace par la suite.

**Parcourir un arbre**, c'est visiter tous ses noeuds, afin de pouvoir opérer une action tour à tour sur eux. Un *parcours d'arbre* définit dans quel ordre les noeuds sont visités.

### Activité 1 : Parcours en largeur d'abord (BFS)

BFS : Breadth First Search

Le parcours en largeur d'abord est un parcours étage par étage (de haut en bas) et de gauche à droite.



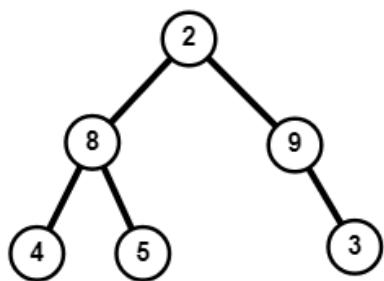
L'ordre des lettres parcourues est donc T-Y-O-P-H-N.

Un parcours en largeur d'abord n'est pas récursif.

### Exercice 1 :

Enoncé

Indiquez dans quel ordre les noeuds sont explorés dans le cas d'un parcours en largeur de l'arbre A1 suivant.



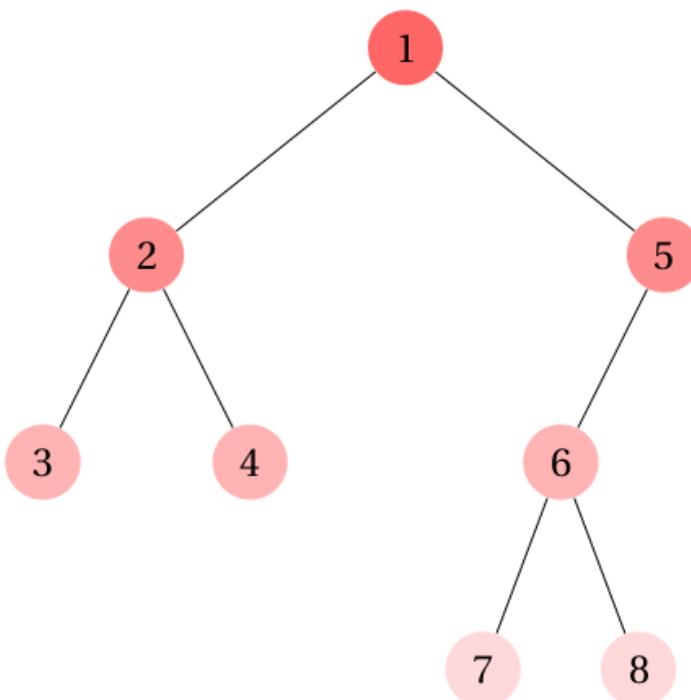
Solution

Dans le cas d'un parcours de cet AB en largeur d'abord, les noeuds sont visités dans l'ordre : 2 - 8 - 9 - 4 - 5 - 3.

### Exercice 2 :

Enoncé

Indiquez dans quel ordre les noeuds sont explorés dans le cas d'un parcours en largeur de l'arbre A2 suivant.



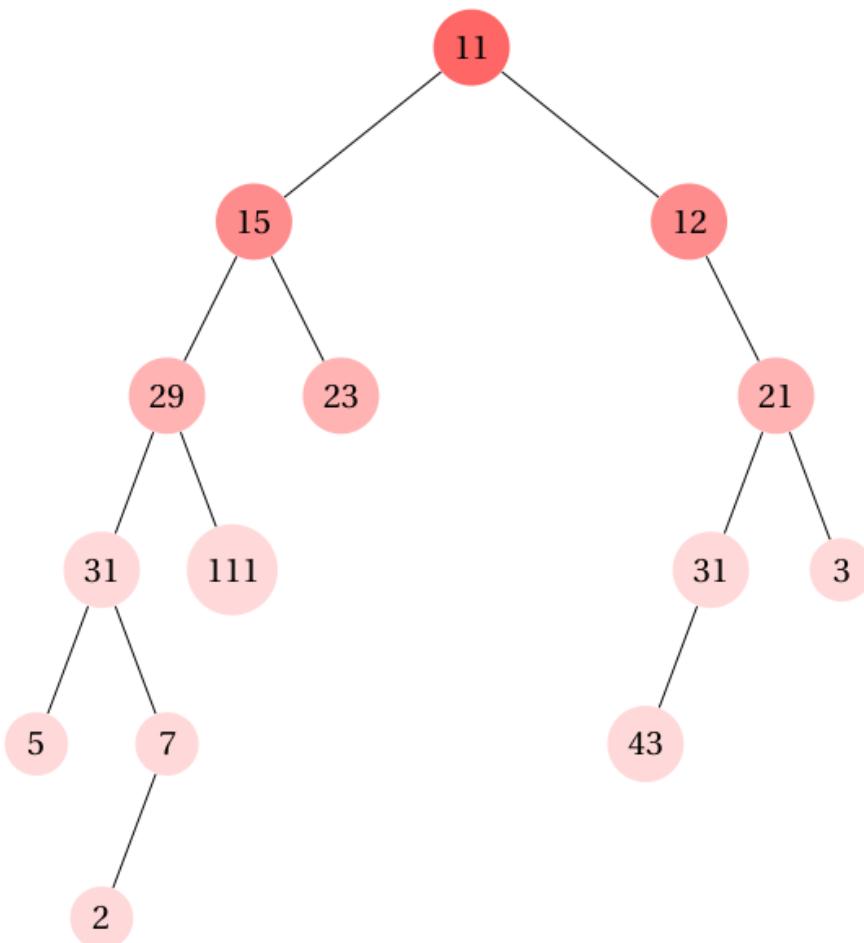
Solution

1 - 2 - 5 - 3 - 4 - 6 - 7 - 8

**Exercice 3 :**

Enoncé

Indiquez dans quel ordre les noeuds sont explorés dans le cas d'un parcours en largeur de l'arbre A3 suivant.

**Solution**

11 - 15 - 12 - 29 - 23 - 21 - 31 - 111 - 31 - 3 - 5 - 7 - 43 - 3

**Algorithme de parcours en largeur**

L'utilisation d'une **file** permet d'écrire facilement l'algorithme de parcours en largeur d'abord. Le principe est le suivant :

- On enfile l'arbre de départ
- Tant que la file n'est pas vide :
  - on défile un élément
  - si celui-ci n'est pas un arbre vide :
    - on affiche son étiquette
    - on enfile ses fils gauche et droit (dont les racines sont les noeuds du niveau suivant)

On utilise ainsi la file pour y insérer et donc traiter (en défilant) tour à tour les noeuds, niveau par niveau.

On importera l'objet `Queue()` du module `queue` de Python, qui permet de : - créer une file vide avec `file = Queue()` - défiler un élément par `file.get()` - enfiler l'élément `a` par `file.put(a)` - savoir si la file est vide par le booléen `file.empty()`

et la classe `Arbre` pour représenter un AB :

```

class Arbre:
    def __init__(self,valeur):
        """Initialisation de l'arbre racine + sous-arbre gauche et sous-arbre droit"""
        self.v=valeur
        self.gauche=None
        self.droit=None

    def ajout_gauche(self,val):
        """On ajoute valeur dans le sous-arbre gauche sous la forme [val,None,None]"""
        self.gauche=Arbre(val)

    def ajout_droit(self,val):
        """ On ajoute valeur dans le sous-arbre droit sous la forme [val,None,None]"""
        self.droit=Arbre(val)

    def affiche(self):
        """permet d'afficher un arbre"""
        if self==None:
            return None
        else :
            return [self.v,Arbre.affiche(self.gauche),Arbre.affiche(self.droit)]

    def taille(self):
        if self==None:
            return 0
        else :
            return 1+Arbre.taille(self.droit)+Arbre.taille(self.gauche)

    def hauteur(self):
        if self==None:
            return 0
        elif self.gauche==None and self.droit==None:
            return 0
        else :
            return 1+max(Arbre.hauteur(self.gauche),Arbre.hauteur(self.droit))

    def get_gauche(self):
        return self.gauche

    def get_droit(self):
        return self.droit

    def get_valeur(self):
        if self==None:
            return None
        else:
            return self.v

```

```

from queue import Queue

def BFS(arbre):
    file = Queue()
    file.put(arbre)
    sol = []
    while file.empty() is False :
        a = file.get()
        if a is not None :
            sol.append(a.get_valeur())
            file.put(a.get_gauche())
            file.put(a.get_droit())
    return sol

```

```

a = Arbre(4)
a.ajout_gauche(3)
a.ajout_droit(1)
a.droit.ajout_gauche(2)
a.droit.ajout_droit(7)
a.gauche.ajout_gauche(6)

```

```
a.droit.droit.ajout_gauche(9)
print(a.affiche())
```

```
[4, [3, [6, None, None], None], [1, [2, None, None], [7, [9, None, None], None]]]
```

```
BFS(a)
```

```
[4, 3, 1, 6, 2, 7, 9]
```

## Activité 2 : Parcours en profondeur

Dans le cas où on explore complètement l'un des deux sous-arbres avant le second on parle d'un **parcours en profondeur**. On utilise le terme *profondeur* car dans ce cas on tente toujours de visiter le noeud le plus éloigné de la racine à condition qu'il soit le fils d'un noeud déjà visité.

On distingue trois ordres particuliers pour explorer en profondeur les sous-arbres gauche, droit et la racine du noeud courant :

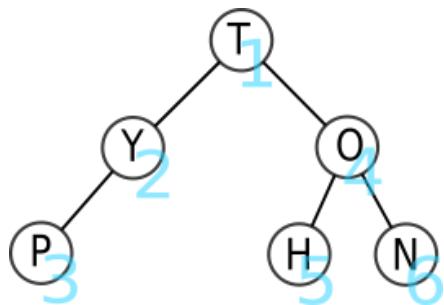
- **ordre préfixe** : le noeud courant est traité, puis son sous-arbre gauche et son sous-arbre droit.
- **ordre infixé** : le noeud courant est traité entre son sous-arbre gauche et son sous-arbre droit.
- **ordre suffixe** : le noeud courant est traité après son sous-arbre gauche et son sous-arbre droit.

### Parcours préfixe

Le parcours **préfixe** est un parcours **en profondeur d'abord**.

**Méthode du parcours préfixe** : (parfois aussi appelé *préordre*)

- Chaque noeud est visité avant que ses fils le soient.
- On part de la racine, puis on visite son fils gauche (et éventuellement le fils gauche de celui-ci, etc.) avant de remonter et de redescendre vers le fils droit.



L'ordre des lettres parcourues est donc T-Y-P-O-H-N.

[Vidéo explicative d'un parcours préfixe](#)

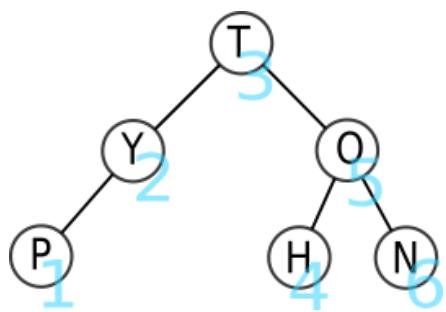
### Parcours infixé

Le parcours **infixé** est aussi un parcours en profondeur d'abord.

**Méthode du parcours infixé** : (parfois aussi appelé *en ordre*)

- Chaque noeud est visité **après son fils gauche mais avant son fils droit**.

- On part donc de la feuille la plus à gauche et on remonte par vagues successives. Un nœud ne peut pas être visité si son fils gauche ne l'a pas été.



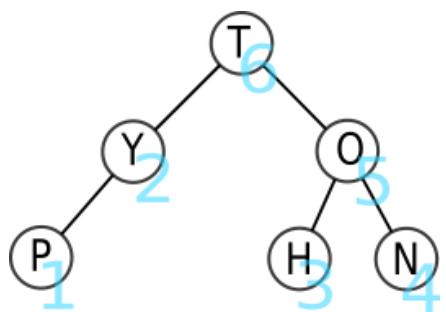
L'ordre des lettres parcourues est donc P-Y-T-H-O-N.

[Vidéo explicative d'un parcours infixé](#)

### Parcours suffixe

Le parcours **suffixe** est aussi un parcours en profondeur d'abord.

**Méthode du parcours suffixe** : (parfois aussi appelé *post ordre*) - Chaque nœud est visité **après ses fils le soient**. - On part donc de la feuille la plus à gauche, et on ne remonte à un nœud père que si ses fils ont tous été visités.



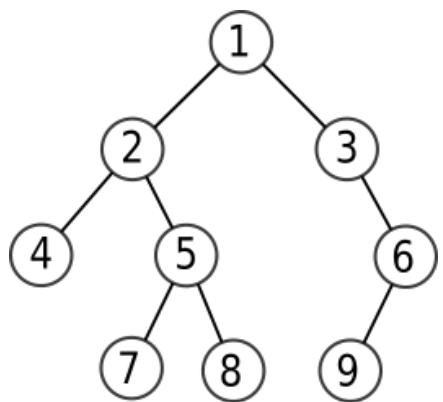
L'ordre des lettres parcourues est donc P-Y-H-N-O-T.

### Exercice n°4 :

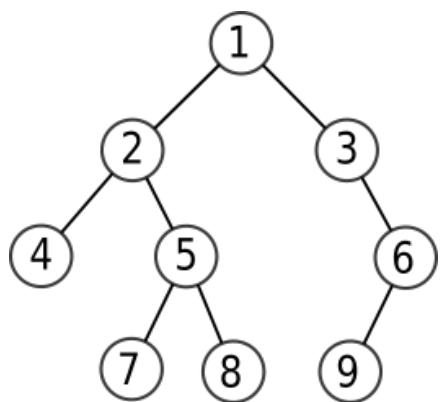
Enoncé

Donner le rendu de chaque parcours :

1. Parcours en largeur
2. Parcours préfixe
3. Parcours infixé
4. Parcours suffixe



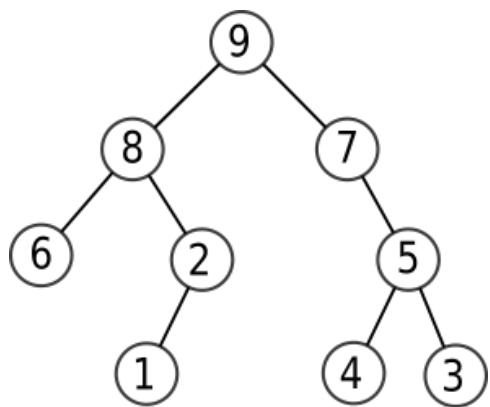
Solution



1. largeur : 1 2 3 4 5 6 7 8 9
2. préfixe : 1 2 4 5 7 8 3 6 9
3. infixé : 4 2 7 5 8 1 3 9 6
4. suffixe : 4 7 8 5 2 9 6 3 1

Exercice n°5 :

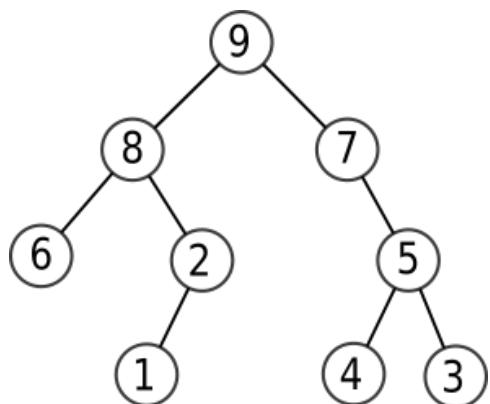
Enoncé



Donner le rendu de chaque parcours :

1. Parcours en largeur
2. Parcours préfixe
3. Parcours infixe
4. Parcours suffixe

Solution



1. largeur : 9 8 7 6 2 5 1 4 3
2. préfixe : 9 8 6 2 1 7 5 4 3
3. infixe : 6 8 1 2 9 7 4 5 3
4. suffixe : 6 1 2 8 4 3 5 7 9

Utilisation de l'implémentation : parcours

**Rappel de l'implémentation :**

```

class Arbre:
    def __init__(self, valeur):
        """Initialisation de l'arbre racine + sous-arbre gauche et sous-arbre droit"""
        self.v=valeur
  
```

```

        self.gauche=None
        self.droit=None

    def ajout_gauche(self,val):
        """On ajoute valeur dans le sous-arbre gauche sous la forme [val,None,None]"""
        self.gauche=Arbre(val)

    def ajout_droit(self,val):
        """ On ajoute valeur dans le sous-arbre droit sous la forme [val,None,None]"""
        self.droit=Arbre(val)

    def affiche(self):
        """permet d'afficher un arbre"""
        if self==None:
            return None
        else :
            return [self.v,Arbre.affiche(self.gauche),Arbre.affiche(self.droit)]

    def taille(self):
        if self==None:
            return 0
        else :
            return 1+Arbre.taille(self.droit)+Arbre.taille(self.gauche)

    def hauteur(self):
        if self==None:
            return 0
        elif self.gauche==None and self.droit==None:
            return 0
        else :
            return 1+max(Arbre.hauteur(self.gauche),Arbre.hauteur(self.droit))

    def get_gauche(self):
        return self.gauche

    def get_droit(self):
        return self.droit

    def get_valeur(self):
        if self==None:
            return None
        else:
            return self.v

```

### Parcours prefixe

```

def parcours_prefixe(arbre):

    if arbre==None:
        return None
    else :
        print(arbre.get_valeur())
        parcours_prefixe(arbre.gauche)
        parcours_prefixe(arbre.droit)

```

**Question :** Implémenter l'arbre de l'exercice 5, puis tester les divers parcours en profondeur.

```

a = Arbre(4)
a.ajout_gauche(3)
a.ajout_droit(1)
a.droit.ajout_gauche(2)
a.droit.ajout_droit(7)
a.gauche.ajout_gauche(6)
a.droit.droit.ajout_gauche(9)
print(a.affiche())

```

```
[4, [3, [6, None, None], None], [1, [2, None, None], [7, [9, None, None], None]]]
```

```
parcours_prefixe(a)
```

```
4  
3  
6  
1  
2  
7  
9
```

### Parcours infixé

```
def infixe(arbre):  
    if arbre is None :  
        return 0  
    infixe(arbre.gauche)  
    print(arbre.v, end = '-')  
    infixe(arbre.droit)
```

```
infixe(a)
```

```
6-3-4-2-1-9-7-
```

### Parcours suffixé

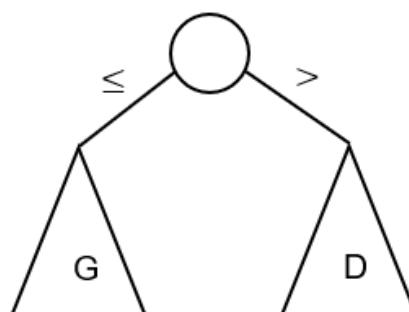
```
def postfixe(arbre):  
    if arbre is None :  
        return 0  
    postfixe(arbre.gauche)  
    postfixe(arbre.droit)  
    print(arbre.v, end = '-')
```

```
postfixe(a)
```

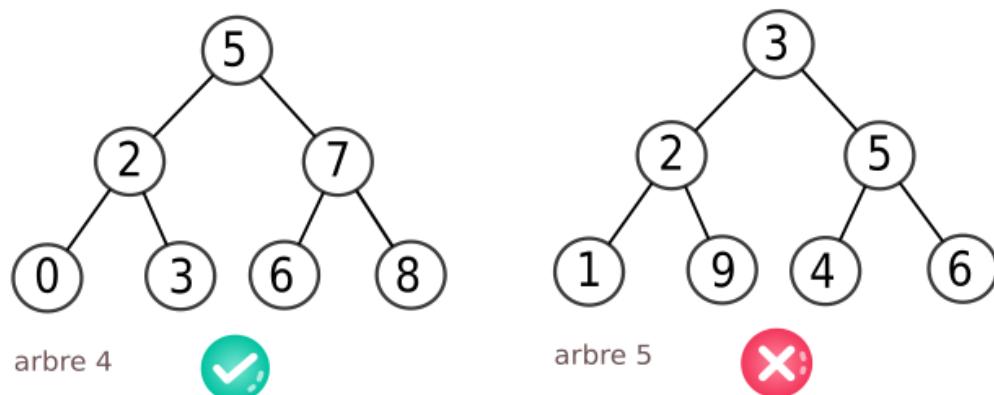
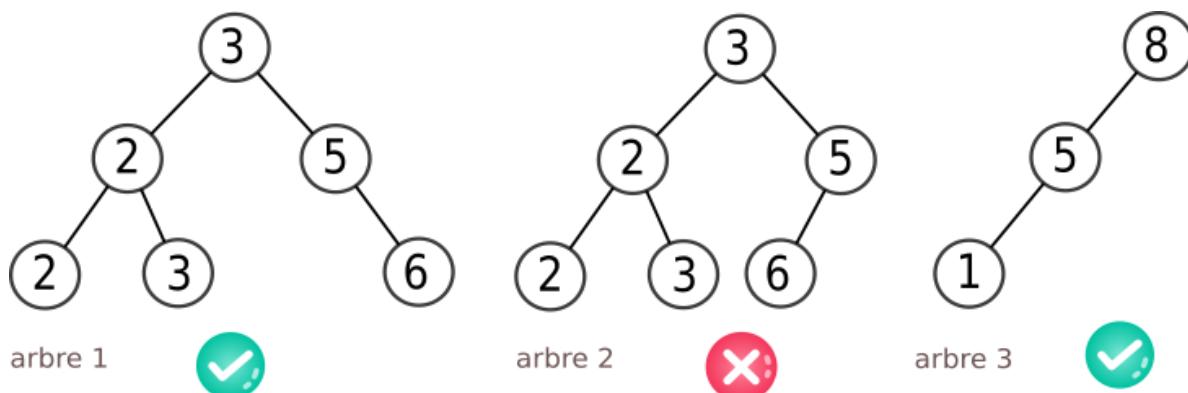
```
6-3-2-9-7-1-4-
```

## V. Arbres binaires de recherche (ABR)

Un **arbre binaire de recherche** est un arbre binaire dont les valeurs des nœuds (valeurs qu'on appelle étiquettes, ou clés) vérifient la propriété suivante : - l'étiquette d'un nœud est **supérieure ou égale** à celle de **chaque** nœud de son **sous-arbre gauche**. - l'étiquette d'un nœud est **strictement inférieure** à celle du **chaque** nœud de son **sous-arbre droit**.



Exemple :



À noter que l'arbre 3 (qui est bien un ABR) est appelé **arbre filiforme**.

L'arbre 5 n'est pas un ABR à cause de la feuille 9, qui fait partie du sous-arbre gauche de 3 sans lui être inférieure.

**Remarque :** on pourrait aussi définir un ABR comme un arbre dont le parcours infixé est une suite croissante.

Exercice n°7 :

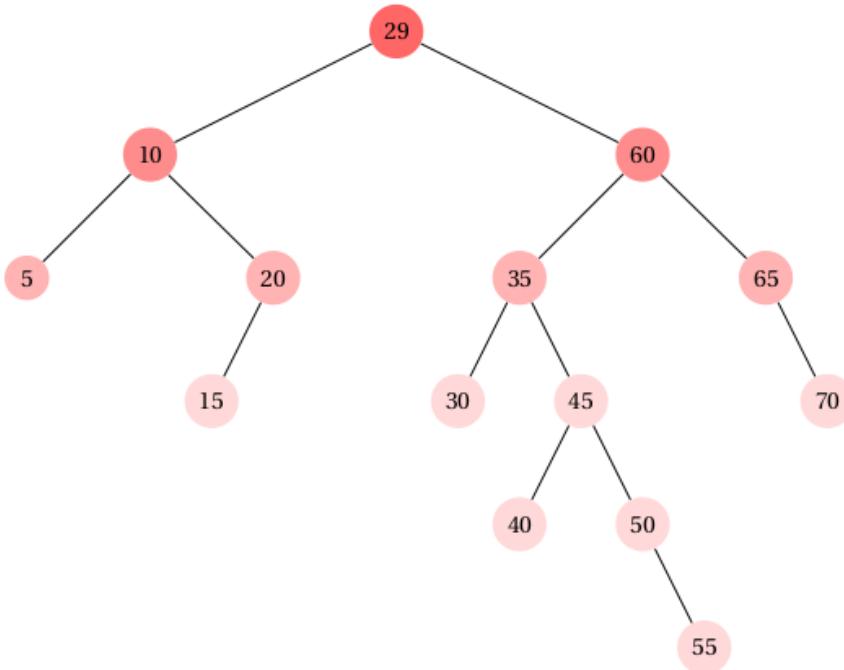
Enoncé

On donne ci-dessous une liste de nombres aléatoires de 14 nombres entiers :

|25 |60 |35 |10 |5 |20 |65 |45 |70 |40 |50 |55 |30 |15 |

Construire (dans l'ordre de la liste) l'arbre binaire de recherche associé.

Solution



Exercice n°8 :

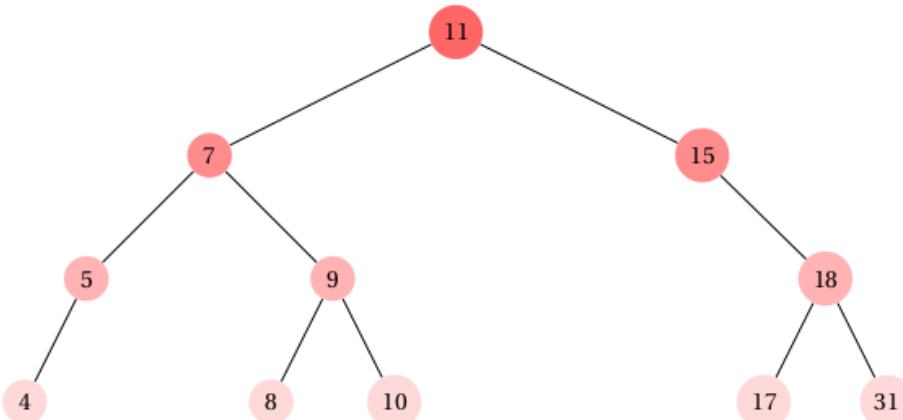
Enoncé

On donne ci-dessous une liste de nombre aléatoire de 11 nombres entiers :

|11 |15 |7 |18 |5 |9 |17 |8 |10 |31 |4 |

Construire (dans l'ordre de la liste) l'arbre binaire de recherche associé.

Solution



Déterminer si un arbre est un ABR

Employer une méthode récursive imposerait de garder en mémoire dans l'exploration des sous-arbres la valeur maximale ou minimale. Nous allons plutôt utiliser la remarque précédente, et nous servir du parcours infixé.

Méthode : récupérer le parcours infixé dans une liste, et faire un test sur cette liste.

```
def est_ABR(arbre, p):
    '''renvoie un booléen indiquant si arbre est un ABR'''
    # p est la liste qui contiendra le parcours. la fonction est à appeler par est_ABR(a, [])
    if arbre is None :
        return True
    est_ABR(arbre.gauche, p)
    p.append(arbre.v)
    est_ABR(arbre.droit, p)
    return p == sorted(p) # on regarde si le parcours est égal au parcours trié
```

```
a = Arbre(5)
a.ajout_gauche(2)
a.ajout_droit(7)
a.gauche.ajout_gauche(0)
a.gauche.ajout_droit(3)
a.droit.ajout_gauche(6)
a.droit.ajout_droit(8)

est_ABR(a, [])
```

True

## Rechercher une clé dans un ABR

Un arbre binaire de taille  $n$  contient  $n$  clés (pas forcément différentes). Pour savoir si une valeur particulière fait partie des clés, on peut parcourir tous les nœuds de l'arbre, jusqu'à trouver (ou pas) cette valeur dans l'arbre. Dans le pire des cas, il faut donc faire  $n$  comparaisons.

Mais si l'arbre est un ABR, le fait que les valeurs soient «rangées» va considérablement améliorer la vitesse de recherche de cette clé, puisque la moitié de l'arbre restant sera écartée après chaque comparaison.

```
def contient_valeur(arbre, valeur):
    if arbre is None :
        return False
    if arbre.get_valeur() == valeur :
        return True
    if valeur < arbre.get_valeur() :
        return contient_valeur(arbre.gauche, valeur)
    else:
        return contient_valeur(arbre.droit, valeur)
```

contient\_valeur(a, 8)

True

contient\_valeur(a, 18)

False

## VI. Sujets BAC

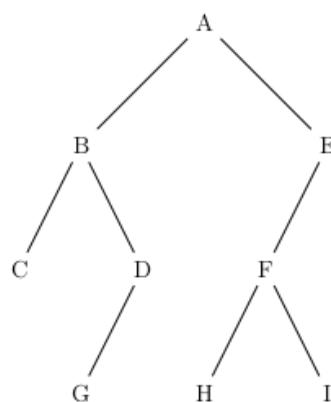
## Sujet n°1 : sujet zéro

Dans cet exercice, on utilisera la convention suivante : la hauteur d'un arbre binaire ne comportant qu'un noeud est 1.

### "Question 1

Enoncé

Déterminer la taille et la hauteur de l'arbre binaire suivant :



Solution

Taille = 9 et hauteur = 4

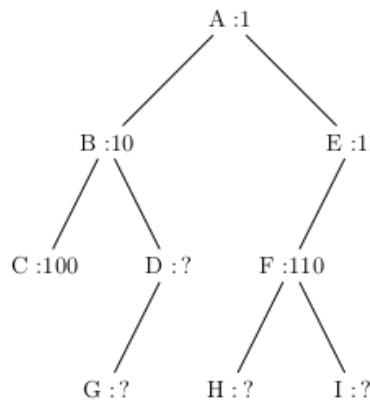
### Question 2

Enoncé

On décide de numérotier en binaire les noeuds d'un arbre binaire de la façon suivante :

- la racine correspond à 1 ;
- la numérotation pour un fils gauche s'obtient en ajoutant le chiffre 0 à droite au numéro de son père ;
- la numérotation pour un fils droit s'obtient en ajoutant le chiffre 1 à droite au numéro de son père ;

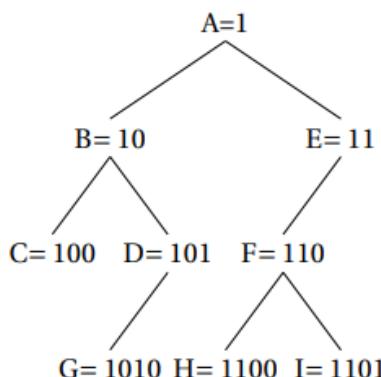
Par exemple, dans l'arbre ci-dessous, on a utilisé ce procédé pour numérotier les noeuds A; B; C; E et F



1. Dans l'exemple précédent, quel est le numéro en binaire associé au noeud G?
2. Quel est le noeud dont le numéro en binaire vaut 13 en décimal ?
3. En notant  $h$  la hauteur de l'arbre, sur combien de bits seront numérotés les noeuds les plus en bas ?
4. Justifier que pour tout arbre de hauteur  $h$  et de taille  $n \geq 2$ , on a :  $h \leq n \leq 2^h - 1$

Solution 2.1

2.1 G : 1010



Solution 2.2

2.2 Noeud I

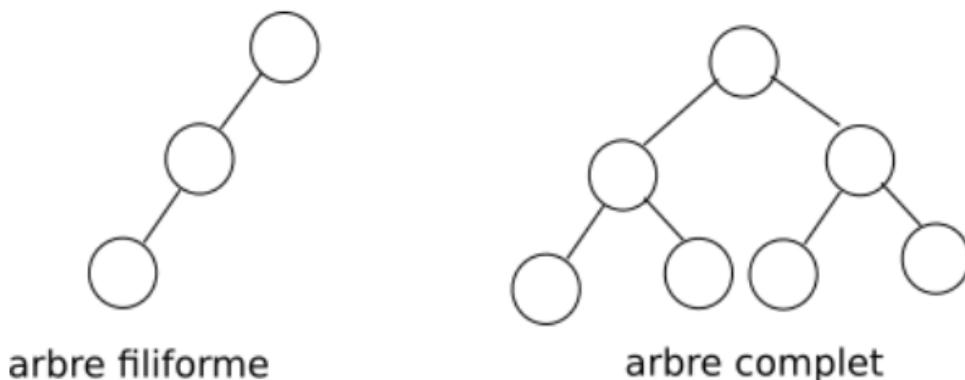
Solution 2.3

2.3 À chaque "étage", on augmente le nombre de bits de 1 : si  $h = 1$ , nombre de bit = 1 ; si  $h = 2$ , nombre de bits = 2... pour une hauteur  $h$  le nombre de bits est de  $h$

Solution 2.4

2.4 Prenons un exemple avec  $h = 3$  : nous avons 2 cas extrêmes : un arbre filiforme ou un arbre complet. Toutes les autres possibilités sont des cas intermédiaires.

un arbre complèt. Toutes les autres possibilités sont des cas intermédiaires.



Dans le cas de l'arbre filiforme nous avons, pour  $h = 3$ ,  $n = 3$  ( $n$  : taille). Si on généralise pour un arbre de hauteur  $h$ , nous avons  $n = h$

Dans le cas d'un arbre complet, pour  $h = 3$  nous avons  $n = 7$ , donc  $n = 2^3 - 1 = 7$ .

Si on généralise pour un arbre de hauteur  $h$ , nous avons  $n = 2^h - 1$

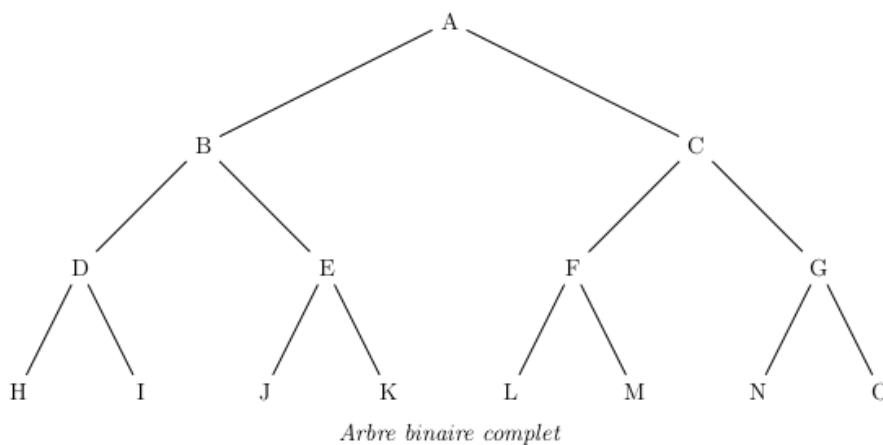
Sachant qu'un arbre quelconque est un intermédiaire entre l'arbre filiforme et l'arbre complet, nous pouvons donc dire que :

$$h \leq n \leq 2^h - 1$$

### Question 3

Enoncé

Un arbre binaire est dit complet si tous les niveaux de l'arbre sont remplis.



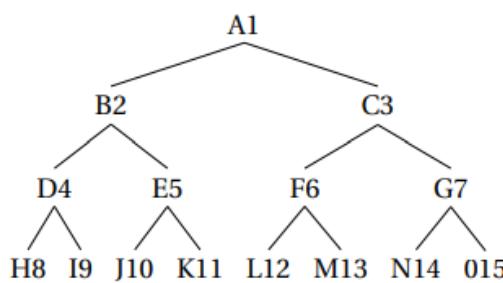
On décide de représenter un arbre binaire complet par un tableau de taille  $n + 1$ , où  $n$  est la taille de l'arbre, de la façon suivante :

- La racine a pour indice 1 ;
- Le fils gauche du noeud d'indice  $i$  a pour indice  $2 \times i$  ;
- Le fils droit du noeud d'indice  $i$  a pour indice  $2 \times i + 1$  ;

- On place la taille  $n$  de l'arbre dans la case d'indice 0.

- 1) Déterminer le tableau qui représente l'arbre binaire complet de l'exemple précédent.
- 2) On considère le père du noeud d'indice  $i$  avec  $i \geq 2$ . Quel est son indice dans le tableau ?

Solution 3.1



[15, 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']

Solution 3.2

Le père d'un fils d'indice  $i$  a pour indice  $i/2$  si  $i$  est pair et  $(i - 1)/2$  sinon.

Sous python on peut dans ce cas utiliser la fonction `//`.

`a//b` donne le quotient de la division euclidienne de `a` par `b`.

#### Question 4

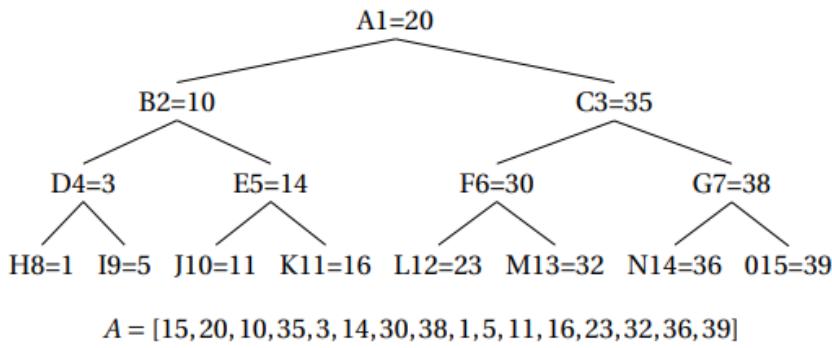
Enoncé

On se place dans le cas particulier d'un arbre binaire de recherche complet où les noeuds contiennent des entiers et pour lequel la valeur de chaque noeud est supérieure à celles des noeuds de son fils gauche, et inférieure à celles des noeuds de son fils droit.

Écrire une fonction recherche ayant pour paramètres un arbre arbre et un élément element. Cette fonction renvoie True si element est dans l'arbre et False sinon. L'arbre sera représenté par un tableau comme dans la question précédente.

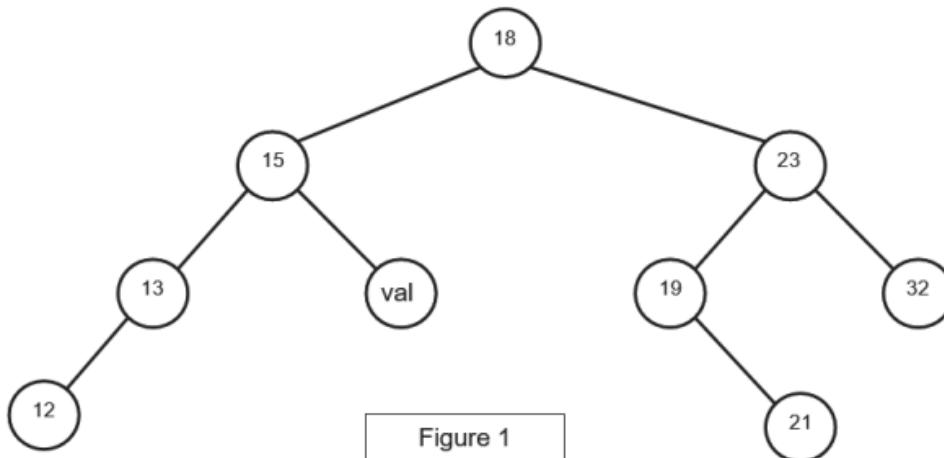
Solution

On a par exemple un arbre de ce type :



```
def recherche(arbre, element):
    '''In : arbre et element entier
    Out : true si element est dans la liste'''
    taille = arbre[0]
    i=1
    while i<=taille:
        if element==arbre[i]:
            return True
        elif element>arbre[i]:
            i=2*i+1
        else:
            i=2*i
    return False
```

Dans cet exercice, les arbres binaires de recherche ne peuvent pas comporter plusieurs fois la même clé. De plus, un arbre binaire de recherche limité à un noeud a une hauteur de 1. On considère l'arbre binaire de recherche représenté ci-dessous (figure 1), où val représente un entier :



### Question 1

Enoncé

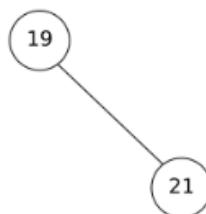
- Donner le nombre de feuilles de cet arbre et préciser leur valeur (étiquette).
- Donner le sous arbre-gauche du noeud 23.

- c. Donner la hauteur et la taille de l'arbre.
- d. Donner les valeurs entières possibles de val pour cet arbre binaire de recherche.

solution 1.a

4 feuilles : 12 ; val ; 21 ; 32

Solution 1.b



Solution 1.c

hauteur = 4 ; taille = 9

Solution 1.d

16 ou 17

On suppose, pour la suite de cet exercice, que val est égal à 16.

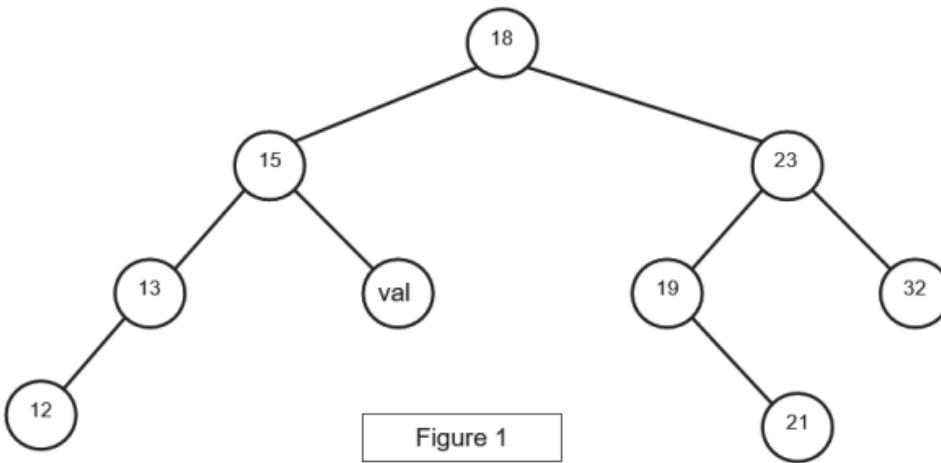
## Question 2

Enoncé

On rappelle qu'un parcours infixé depuis un noeud consiste, dans l'ordre, à faire un parcours infixé sur le sous arbre-gauche, afficher le noeud puis faire un parcours infixé sur le sous-arbre droit. Dans le cas d'un parcours suffixe, on fait un parcours suffixe sur le sous-arbre gauche puis un parcours suffixe sur le sous-arbre droit, avant d'afficher le noeud.

- a. Donner les valeurs d'affichage des noeuds dans le cas du parcours infixé de l'arbre.
- b. Donner les valeurs d'affichage des noeuds dans le cas du parcours suffixe de l'arbre.

Solution 2



infixe : 12 – 13 – 15 – 16 – 18 – 19 – 21 – 23 – 32

On considère la classe Noeud définie de la façon suivante en Python :

```
infixe = 12 13 15 16 18 19 21 23 32 20
```

```
class Noeud():
    def __init__(self, v):
        self.ag = None
        self.ad = None
        self.v = v

    def insere(self, v):
        n = self
        est_insere = False
        while not est_insere :
            if v == n.v:
                est_insere = True           | bloc 1
            elif v < n.v:
                if n.ag != None:
                    n = n.ag
                else:
                    n.ag = Noeud(v)
                    est_insere = True      | bloc 2
            else:
                if n.ad != None:
                    n = n.ad
                else:
                    n.ad = Noeud(v)
                    est_insere = True      | bloc 3

    def insere_tout(self, vals):
        for v in vals:
            self.insere(v)
```

### Question 3

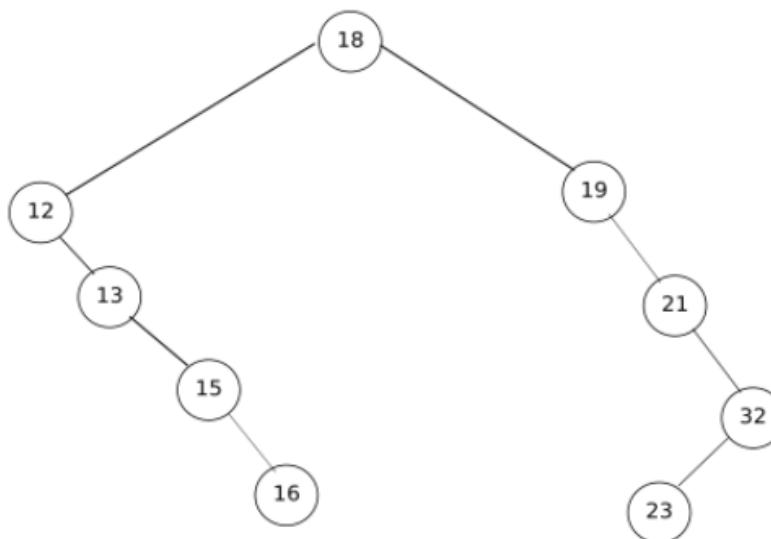
Enoncé

- a. Représenter l'arbre construit suite à l'exécution de l'instruction suivante :

```
racine = Noeud(18)
racine.insere_tout([12, 13, 15, 16, 19, 21, 32, 23])
```

- b. Ecrire les deux instructions permettant de construire l'arbre de la figure 1. On rappelle que le nombre val est égal à 16.
- c. On considère l'arbre tel qu'il est présenté sur la figure 1. Déterminer l'ordre d'exécution des blocs (repérés de 1 à 3) suite à l'application de la méthode insere(19) au noeud racine de cet arbre.

Solution 3.a



Solution 3.b

```

racine = Noeud(18)
racine.insere_tout([15, 23, 13, 16, 12, 19, 21, 32])
  
```

Solution 3.c

bloc 3 – bloc 2 – bloc 1

#### Question 4

Enoncé

Ecrire une méthode recherche(self, v) qui prend en argument un entier v et renvoie la valeur True si cet entier est une étiquette de l'arbre, False sinon.

Solution

```

def recherche (self,v):
    n = self
    while n is not None:
        if v < n.v:
            n = n.ag
        elif v > n.v:
  
```

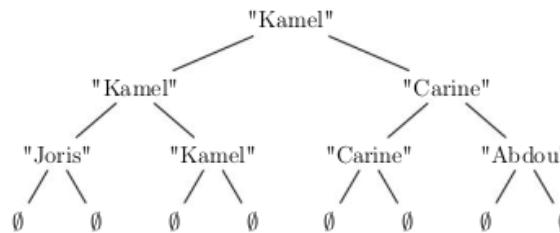
```

n = n.ad
else:
    return True
return False

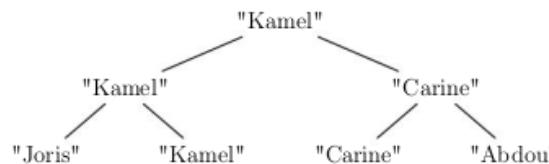
```

### Sujet n°3

La fédération de badminton souhaite gérer ses compétitions à l'aide d'un logiciel. Pour ce faire, une structure arbre de compétition a été définie récursivement de la façon suivante : un arbre de compétition est soit l'arbre vide, noté  $\emptyset$ , soit un triplet composé d'une chaîne de caractères appelée valeur, d'un arbre de compétition appelé sous-arbre gauche et d'un arbre de compétition appelé sous-arbre droit. On représente graphiquement un arbre de compétition de la façon suivante :



Pour alléger la représentation d'un arbre de compétition, on ne notera pas les arbres vides, l'arbre précédent sera donc représenté par l'arbre A suivant :



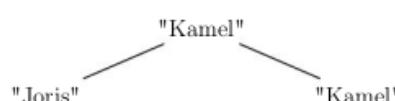
Cet arbre se lit de la façon suivante :

- 4 participants se sont affrontés : Joris, Kamel, Carine et Abdou. Leurs noms apparaissent en bas de l'arbre, ce sont les valeurs de feuilles de l'arbre.
- Au premier tour, Kamel a battu Joris et Carine a battu Abdou.
- En finale, Kamel a battu Carine, il est donc le vainqueur de la compétition.

Pour s'assurer que chaque finaliste ait joué le même nombre de matchs, un arbre de compétition a toutes ces feuilles à la même hauteur.

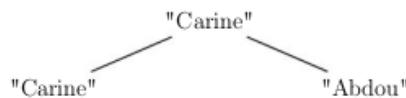
Les quatre fonctions suivantes pourront être utilisées :

- La fonction racine qui prend en paramètre un arbre de compétition arb et renvoie la valeur de la racine.  
Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, racine(A) vaut "Kamel".
- La fonction gauche qui prend en paramètre un arbre de compétition arb et renvoie son sousarbre gauche.  
Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, gauche(A) vaut l'arbre représenté graphiquement ci-après :



- La fonction droit qui prend en argument un arbre de compétition arb et renvoie son sous-arbre droit.  
Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, droit(A) vaut l'arbre représenté

graphiquement ci-dessous :



— La fonction `est_vide` qui prend en argument un arbre de compétition et renvoie True si l'arbre est vide et False sinon.

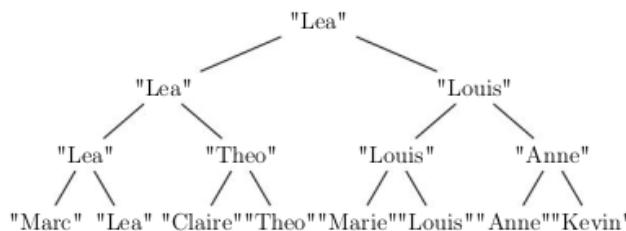
Exemple : en reprenant l'exemple d'arbre de compétition présenté ci-dessus, `est_vide(A)` vaut False

Pour toutes les questions de l'exercice, on suppose que tous les joueurs d'une même compétition ont un prénom différent.

### Question 1

Enoncé

(a) On considère l'arbre de compétition B suivant :



Indiquer la racine de cet arbre puis donner l'ensemble des valeurs des feuilles de cet arbre.

(b) Proposer une fonction Python vainqueur prenant pour argument un arbre de compétition arb ayant au moins un joueur. Cette fonction doit renvoyer la chaîne de caractères constituée du nom du vainqueur du tournoi.

Exemple : `vainqueur(B)` vaut "Lea"

(c) Proposer une fonction Python finale prenant pour argument un arbre de compétition arb ayant au moins deux joueurs. Cette fonction doit renvoyer le tableau des deux chaînes de caractères qui sont les deux compétiteurs finalistes.

Exemple : `finale(B)` vaut ["Lea", "Louis"]

Solution 1.(a)

racine => "Lea"

feuilles => "Marc", "Lea", "Claire", "Theo", "Marie", "Louis", "Anne" et "Kevin"

Solution 1.(b)

```

def vainqueur(arb):
    return racine(arb)

def finale(arb):
    f1 = gauche(arb)
    f2 = droit(arb)
    return [racine(f1), racine(f2)]

```

## Question 2

Enoncé

- (a) Proposer une fonction Python occurrences ayant pour paramètre un arbre de compétition arb et le nom d'un joueur nom et qui renvoie le nombre d'occurrences (d'apparitions) du joueur nom dans l'arbre de compétition arb.

Exemple : occurrences(B, "Anne") vaut 2.

- (b) Proposer une fonction Python a\_gagne prenant pour paramètres un arbre de compétition arb et le nom d'un joueur nom et qui renvoie le booléen True si le joueur nom a gagné au moins un match dans la compétition représenté par l'arbre de compétition arb.

Exemple : a\_gagne(B,"Louis") vaut True

Solution 2.(a)

```

def occurrences(arb, nom):
    if est_vide(arb):
        return 0
    elif racine(arb) == nom:
        res = 1
    else:
        res = 0
    return res + occurrences(gauche(arb), nom) + occurrences(droit(arb), nom)

```

Solution 2.(b)

```

def a_gagne(arb, nom):
    return occurrences(arb,nom) > 1

```

## Question 3

Enoncé

- On souhaite programmer une fonction Python nombre\_matchs qui prend pour arguments un arbre de compétition arb et le nom d'un joueur nom et qui renvoie le nombre de matchs joués par le joueur nom dans la compétition représentée par l'arbre de

compétition arb

Exemple : nombre\_matchs(B,"Lea") doit valoir 3 et nombre\_matchs(B,"Marc") doit valoir 1.

(a) Expliquer pourquoi les instructions suivantes renvoient une valeur erronée. On pourra pour cela identifier le noeud de l'arbre qui provoque une erreur.

```
1 def nombre_matchs (arb ,nom ):
2     """ arbre_competition , str -> int """
3     return occurrences (arb ,nom)
```

(b) proposer une correction pour la fonction nombre\_matchs

Solution 3.(a)

Les instructions proposées renvoient une valeur erronée dans le cas où le paramètre nom correspond au vainqueur du tournoi. En effet, si on considère l'arbre proposé à la question 1a, occurrences(arb, "Lea") renvoie 4 alors que Lea a joué seulement 3 matchs.

Solution 3.(b)

```
def nombre_matchs(arb, nom):
    if vainqueur(arb)==nom :
        return occurrences(arb, nom) - 1
    else :
        return occurrences(arb, nom)
```

Question 4

Enoncé

Recopier et compléter la fonction liste\_joueurs qui prend pour argument un arbre de compétition arb et qui renvoie un tableau contenant les participants au tournoi, chaque nom ne devant figurer qu'une seule fois dans le tableau.

L'opération + à la ligne 8 permet de concaténer deux tableaux.

Exemple : Si L1 = [4, 6, 2] et L2 = [3, 5, 1], l'instruction L1 + L2 va renvoyer le tableau [4, 6, 2, 3, 5, 1]

```
1 def liste_joueurs ( arb ):
2     """ arbre_competition -> tableau """
3     if est_vide (arb ):
4         return ...
5     elif ... and ... :
6         return [ racine (arb )]
7     else :
8         return ...+ liste_joueurs ( droit (arb ))"
```

**Solution**

```
def liste_joueur(arb):
    if est_vide (arb):
        return []
    elif est_vide(gauche(arb)) and est_vide(droit(arb)) :
        return [racine(arb)]
    else :
        return liste_joueurs(gauche(arb)) + liste_joueurs(droit(arb))
```