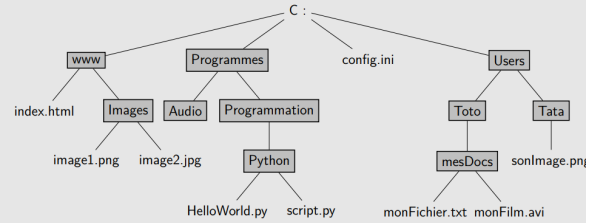
Arbres Binaires : définitions et propriétés

Les listes, piles et files que nous avons croisées jusqu'ici sont utilisées pour représenter de structures pouvant être énumérées séquentiellement. Elles sont particulièrement efficaces lorsqu'il s'agit d'accéder au premier élément (ou au dernier selon l'implémentation). Elles ne le sont pas contre pas quand il s'agit **d'accéder à un élément à une position arbitraire** dans la structure, car il faut parcourir toute la liste/pile/file jusqu'à la position recherchée, ce qui donne un temps d'accès proportionnel à la taille de la structure (donc en).

1. Structures arborescentes

Les **structures arborescentes**, c'est-à-dire sous forme d'arbre, sont une autre forme de **structures chainées** dans laquelle l'accès à un élément se fait potentiellement bien plus rapidement qu'avec les listes chainées.

Ces types de structures arborescentes sont omniprésentes en informatiques, ne serait-ce que par l'organisation du système de fichier :



Structure arborescente

Une **structure arborescente** est une structure chaînée construite à partir d'un point de départ qui se scinde en plusieurs branches à chaque étape.

2. Arbres Binaires

2.1. Définitions et vocabulaire

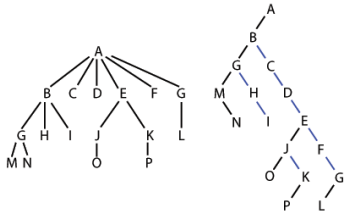
définition : Arbre Binaire

Un **arbre binaire** est un cas particulier de structure arborescente où chaque position ouvre sur exactement deux branches.

Plus précisément, un **arbre binaire** est un ensemble fini de **noeuds** correspondant à l'un des deux cas suivants :

- Soit l'arbre est vide, c'est-à-dire qu'il ne contient aucun **noeud**.
- Soit l'arbre n'est pas vide, et ses **noeuds** sont structurés de la façon suivante :
  - un noeud est appelé **la racine** de l'arbre ;
  - les noeuds restants sont séparés en deux sous-ensembles qui forment récursivement **deux sous-arbres binaires** appelés respectivement **sous-arbre gauche** et **sous-arbre droit** ;
  - la racine est reliée à chacune des racines de ces sous-arbres gauches et droits (à conditions qu'ils ne soient pas vides).

Exemples et contre-exemples d'arbres binaires



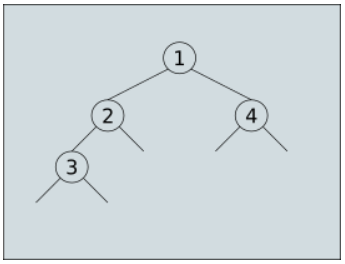
L'arbre de gauche n'est pas un arbre binaire : 6 sous-arbres sont-rattachés à , les sous-arbres de racines

.

L'arbre de droite est bien un arbre binaire, de chaque noeud partent deux sous-arbres, éventuellement vides.

Vocabulaire des arbres

On considère l'arbre binaire ci-dessous :



- La **taille de l'arbre** est , c'est le nombre de noeuds qui le compose.
- Le noeud **racine** est le noeud .
- Le sous-arbre gauche à partir de contient deux noeuds (et), le sous-arbre droit un seul ( ).
- le noeud possède deux **fils** : son **fils gauche** est et son **fils droit** est .
- Le sous-arbre gauche à partir de n'est pas vide (il contient le noeud), le sous-arbre droit lui l'est.
- Le noeud **parent** du noeud est le noeud .
- Les deux sous-arbres à partir de sont vides, touts comme ceux de . On dira que les noeuds et sont des **feuilles** de l'arbre.

Remarques

Les arbres binaires sont utilisés pour traiter des données. Chaque noeud peut donc être représenté par la donnée qu'il contient. Ainsi, dans les arbres ci-dessus :

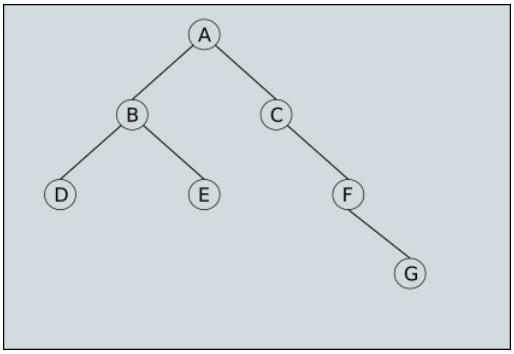
- un contient des valeurs numériques (, , et) ;
- l'autre contient des caractères (à).

Exercice

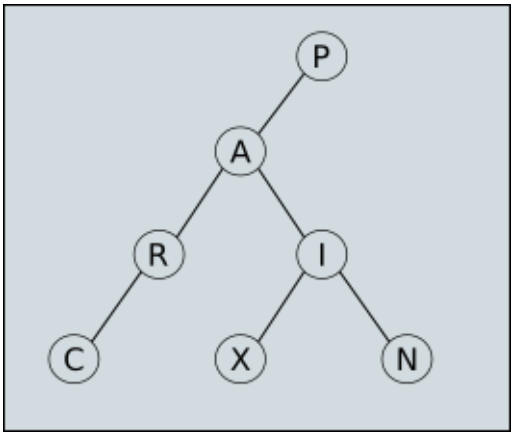
Pour chacun des arbres binaires ci-dessous, préciser sa taille, sa racine ainsi que les noeuds feuilles :

Énoncé

Arbre 1



Arbre 2



Solution

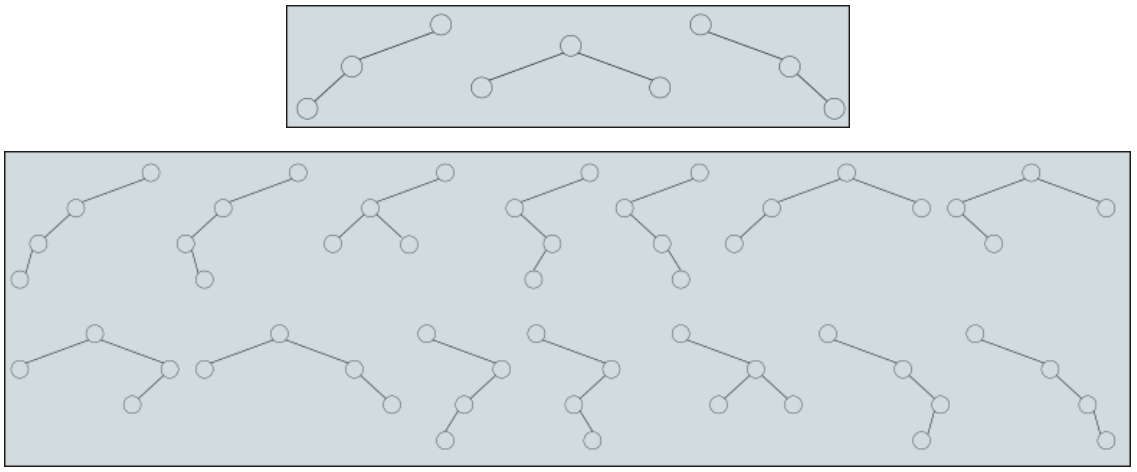
- Arbre 1 :
  - taille : 7
  - racine : « A »
  - feuilles : « D », « E », « G »
- Arbre 2 :
  - taille : 7
  - racine : « P »
  - feuilles : « C », « X », « N »

Exercice

Énoncé

Dessiner tous les arbres binaires ayant respectivement 3 et 4 noeuds.

Solution



Exercice

Énoncé

Sachant qu'il y a 1 arbre binaire vide, 1 arbre binaire contenant 1 noeud, 2 arbres binaires contenant 2 noeuds, 5 arbres binaires contenant 3 noeuds, et 14 arbres binaires contenant 4 noeuds, calculer le nombre d'arbres binaires contenant 5 noeuds, sans chercher à les construire tous.

Solution

Un arbre de taille 5 dispose d'un noeud racine et de 4 noeuds, pouvant être répartis de la manière suivante :

- 4 noeuds dans le sous-arbre gauche, et 0 dans le sous-arbre droit ;
- 3 noeuds dans le sous-arbre gauche, et 1 dans le sous-arbre droit ;
- 2 noeuds dans le sous-arbre gauche, et 2 dans le sous-arbre droit ;
- 1 noeud dans le sous-arbre gauche, et 3 dans le sous-arbre droit ;
- 0 noeud dans le sous-arbre gauche, et 4 dans le sous-arbre droit.

On en déduit que le nombre d'arbres différents à 5 noeuds est :

2.2. Hauteur d'un arbre

Définition : hauteur d'un arbre

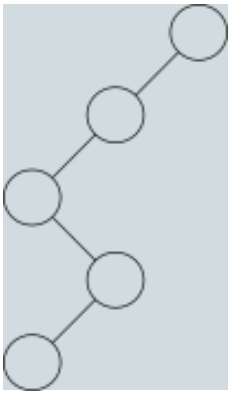
La **hauteur d'un arbre** est égale au nombre maximal de noeuds reliant la racine aux feuilles, les extrémités étant comprises.

Si un arbre est de taille et de hauteur , on a la relation suivante :

Démonstration

Inégalité

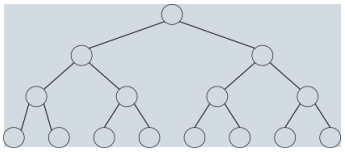
Dans le cas d'un arbre ayant à chaque noeud au moins un de ses sous-arbre vide :



Inégalité

Il est évident que dans ce cas la hauteur de l'arbre est égale à sa taille, d'où .

dans le cas d'un **arbre binaire parfait**, c'est-à-dire dont toutes les feuilles sont situées à la même distance de la racine :



La taille est alors égale à

_____

D'où l'inégalité recherchée.

Hauteur et récursivité

La hauteur d'un arbre peut-aussi être définie récursivement :

- la hauteur d'un arbre vide est 0 ;
- la hauteur d'un arbre est égale à un plus le maximum de la hauteur des deux sous-arbres de la racine :

2.3. Implémentation d'arbres en Python

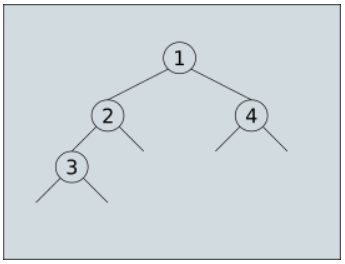
Il existe bien entendu différentes façons d'implémenter une structure d'arbre binaire en Python. Cependant, la méthode la plus simple est d'utiliser le *paradigme Objet* afin de représenter des noeuds :

```
class Node() :
    def __init__(self, valeur, gauche, droit) :
        self.valeur=valeur
        self.gauche = gauche
        self.droit = droit
```

un sous-arbre vide étant représenté par la valeur `None`.

Exemple d'utilisation des objets `Node`

On considère l'arbre binaire ci-dessous :



Une représentation en Python de cet arbre est alors :

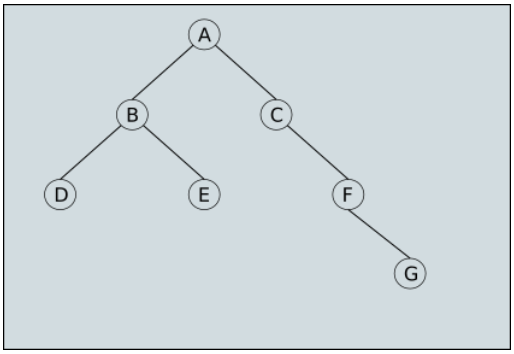
```
tree = Node(1,
        Node(2,
            Node(3, None, None),
            None),
        Node(4, None, None))
```

Exercice

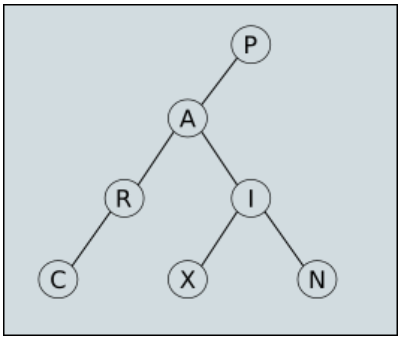
Énoncé

Donner le code de représentation de chacun des arbres ci-dessous en Python :

Arbre 1



Arbre 2



Solution

- Arbre 1 :

```
arbre1 = Node("A",
      Node("B",
            Node("D",None,None),#FG
            Node("E",None,None),#FD
           ),
      Node("C",#FD
            None,#FG
            Node("F",#FD
                 None,#FG
                 Node("G", None, None)#FD
                 )
           )
)
```

- Arbre 2 :

```
arbre2 = Node("P",
      Node("A",
           Node("R",
                Node("C", None, None),
                None
                ),
           Node("I",
                Node("X", None, None),
                Node("N", None, None)
                )
           ),
  None
  )
```

Exercice : Fonction `hauteur`

Énoncé

Coder une fonction `hauteur(t)` calculant la hauteur d'un arbre `t` qui lui est passé en argument (*indice : récursivité*).

Solution

```
def hauteur(t) :
    if t is None :
        return 0
    else :
        return 1 + max(hauteur(t.gauche), hauteur(t.droit))
```

Exercice : Fonction `taille`

Énoncé

Coder une fonction `taille(t)` calculant la taille d'un arbre `t` qui lui est passé en argument (*indice : récursivité*).

Solution

```
def taille(t) :
    if t is None :
        return 0
    else :
        return 1 + taille(t.gauche)+taille(t.droit)
```

Exercice : Fonction `estVide(tree)`

Énoncé

Coder une fonction `estVide(tree)` renvoyant `True` si l'arbre est vide, et `False` sinon.

Solution

```
def estVide(t) :
    return t is None
```

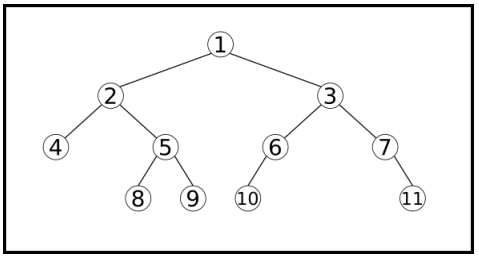
Arbres doublement chainés

Il est parfois - mais pas toujours- nécessaire de mémoriser le noeud parent de chaque noeud. On parle alors d'arbre **doublement chainé**. L'implémentation en POO de la classe `Node` est alors quasiment identique, en rajoutant seulement l'attribut `parent` :

```
class Node() :
    def __init__(self, valeur, gauche, droit, parent) :
        self.valeur=valeur
        self.gauche = gauche
        self.droit = droit
        self.parent = parent
```

3. Parcours d'arbres

Pour utiliser un arbre, il faut le **parcourir**. Or il existe plusieurs ordres de parcours, qui tous ont un intérêt différent. Pour illustrer ces ordres de parcours, nous utiliserons comme exemple le même arbre, dont on veut **afficher les différents noeuds** :



3.1. Parcours en profondeur (DFS ou Depth-First Search)

Les **parcours en profondeur** sont des parcours qui seront traités de manière récursive, en partant de la racine. Il en existe trois types principaux :

Parcours Préfixe

On appelle **parcours préfixe** un parcours où les noeuds seront affichés dans l'ordre suivant

- on affiche la racine ;
- ensuite on affiche récursivement le sous-arbre gauche ;
- enfin on affiche récursivement le sous-arbre droit.

Le parcours est dans l'ordre **noeud - gauche - droit**

Exemple

Parcours Infixe

On appelle **parcours infixe** un parcours où les noeuds seront affichés dans l'ordre suivant

- on affiche récursivement le sous-arbre gauche ;
- ensuite on affiche la racine ;
- enfin on affiche récursivement le sous-arbre droit.

Le parcours est donc dans l'ordre **gauche - noeud -droit**.

Exemple

Parcours Suffixe (ou postfixe)

On appelle **parcours suffixe** un parcours où les noeuds seront affichés dans l'ordre suivant

- on affiche récursivement le sous-arbre gauche ;
- ensuite on affiche récursivement le sous-arbre droit ;
- enfin on on affiche la racine.

Le parcours est donc dans l'ordre **gauche - droit - noeud**.

Exemple

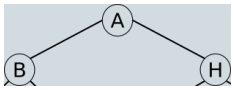
Exercice

Enoncé

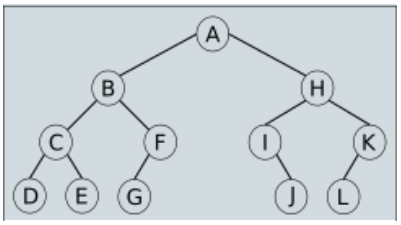
Pour chacun des arbres suivants, donner l'affichage par :

1. un parcours préfixe ;
2. un parcours infixe ;
3. un parcours suffixe.

Arbre 1



Arbre 2



Solution

- Arbre 1 :
  - parcours Préfixe : `A B H`
  - parcours Infixe : `B A H`
  - parcours Suffixe : `B H A`
- Arbre 2 :
  - parcours Préfixe : `A B C D E F G H I J K L`
  - parcours Infixe : `D C E B G F A I J H L K`
  - parcours Suffixe : `D E C G F B J I L K H A`

3.2. Algorithmes et codage des parcours en profondeur

Codage des arbres

Les deux arbres précédents peuvent être codés en Python de la manière suivante :

```
arbre1 = Node('A',
            Node('B', None, None),
            Node('C', None, None)
            )
```

```
arbre2 = Node('A',
            Node('B',
                Node('C',
                    Node('D', None, None),
                    Node('E', None, None)),
                Node('F',
                    Node('G', None, None),
                    None)
                ),
            Node('H',
                Node('I',
                    None,
                    Node('J', None,None)),
                Node('K',
                    Node('L', None,None),
                    None)
                )
            )
```

Exemple : algorithme de parcours préfixe

Un algorithme en langage naturel permettant d'afficher les noeuds d'un arbre par un parcours préfixe peut-être écrit comme suit :

```
visiterPréfixe(Arbre A) :
    Afficher (A)
    Si nonVide (gauche(A))
          visiterPréfixe(gauche(A))
    Si nonVide (droite(A))
          visiterPréfixe(droite(A))
```

Sa traduction en Python est la suivante :

```
def visitePrefixe(tree) :
    print(tree.valeur, end=" ")
    if not(estVide(tree.gauche)) :
        visitePrefixe(tree.gauche)
    if not(estVide(tree.droit)) :
        visitePrefixe(tree.droit)
```

Exercice : Parcours infixe

Enoncé

Créer une fonction `visiteInfixe` permettant d'afficher les noeuds d'un arbre par un parcours infixe. La fonction sera testée sur les deux arbres donnés plus haut.

Solution

```
def visiteInfixe(tree) :

    if not(estVide(tree.gauche)) :
        visiteInfixe(tree.gauche)
    print(tree.valeur, end=" ")
    if not(estVide(tree.droit)) :
        visiteInfixe(tree.droit)
```

Exercice : Parcours suffixe

Enoncé

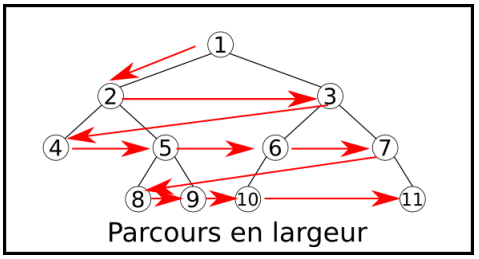
Créer une fonction `visiteSuffixe` permettant d'afficher les noeuds d'un arbre par un parcours suffixe. La fonction sera testée sur les deux arbres donnés plus haut.

Solution

```
def visiteSuffixe(tree) :
    if not(estVide(tree.gauche)) :
        visiteSuffixe(tree.gauche)
    if not(estVide(tree.droit)) :
        visiteSuffixe(tree.droit)
    print(tree.valeur, end=" ")
```

3.3. Parcours en largeur

Le **parcours en largeur** d'un arbre consiste à parcourir **chaque niveau** de l'arbre de gauche à droite, en partant de la racine.



Sur cet arbre, le parcours en largeur affichera les noeuds dans l'ordre suivant : 1 2 3 4 5 6 7 8 9 10 11

Exercice : application directe

Énonce

Donner le résultat d'un parcours en largeur des deux arbres des exercices précédents.

Solution

- Arbre 1 : `A B H`
- Arbre 2 : `A B H C F I K D E G J L`

Algorithme de parcours en largeur

Le parcours en largeur **n'est pas effectué récursivement**. Il fonctionne avec le principe **d'une file** :

```
ParcoursLargeur(Arbre A) {
   f = FileVide
   enfiler(Racine(A), f)
   Tant que (f != FileVide) {
       noeud = defiler(f)
       Afficher(noeud)
       Si (gauche(noeud) != null) Alors
           enfiler(gauche(noeud), f)
       Si (droite(noeud) != null) Alors
           enfiler(droite(noeud), f)
   }
}
```

Exercice

Énoncé

Implémenter une fonction `visiteLargeur(tree)`, utilisant une structure de file basée sur les listes python, et utilisant :

- `list.insert(0, e)` pour enfiler l'élément `e` à la position 0 ;
- `list.pop()` pour défiler le dernier élément de la file.

Solution

```
def visiteLargeur(tree) :
    f = []
    f.insert(0, tree)
    while f!= [] :
        noeud = f.pop()
        print(noeud.valeur, end=" ")
        if noeud.gauche :
            f.insert(0, noeud.gauche)
        if noeud.droit :
            f.insert(0, noeud.droit)
```