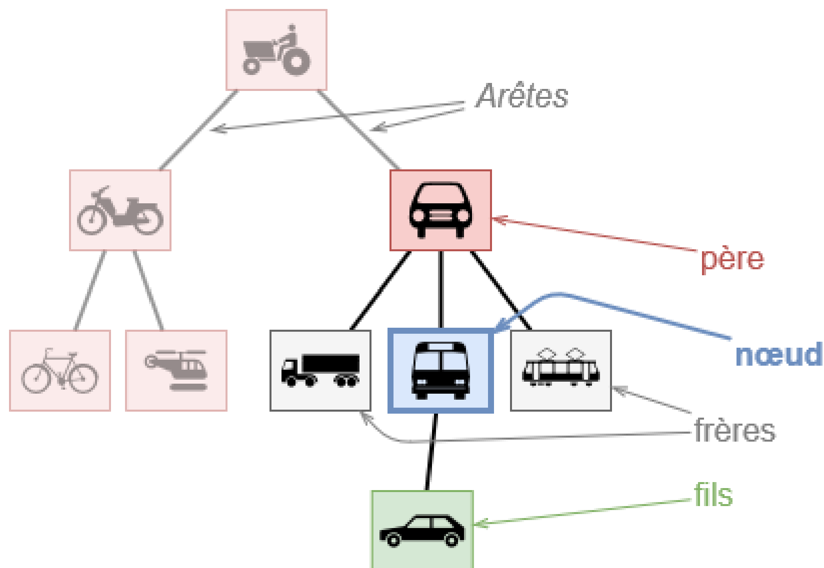
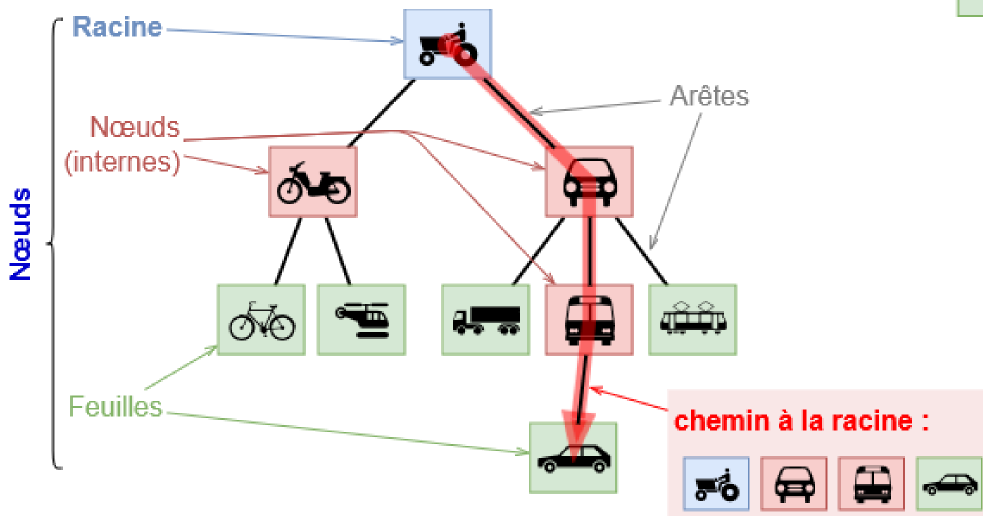
Chapitre 16 - Arbres binaires

Les **arbres** sont des types abstraits très utilisés en informatique, notamment quand on a besoin d'une **structure hiérarchique** des données.

1- Vocabulaire :

Un arbre est constitué de **nœuds**, reliés entre eux par des **arêtes** selon une relations *pères -fils* :





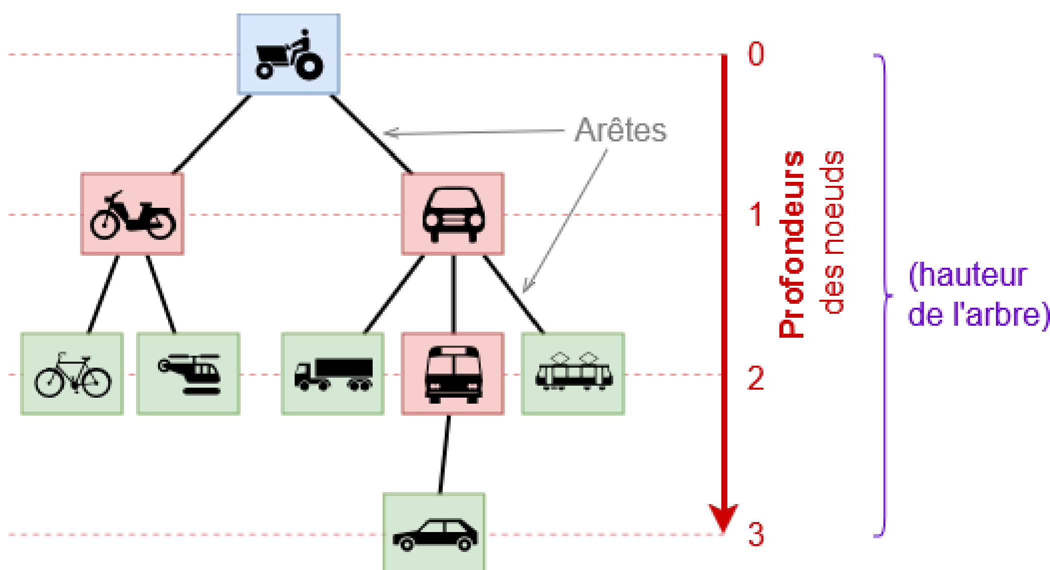
On distingue trois types de **nœuds** :

- La **racine** de l'arbre est l'unique nœud ne possédant <u>pas de parent</u>.
- les **feuilles** (ou *nœuds externes*), éléments ne possédant <u>pas de fils</u> dans l'arbre ;
- les **nœuds internes**, éléments possédant des fils (sous-branches).

La **profondeur** d'un nœud est la distance (<u>nombre d'arêtes</u>) de la racine au nœud.

La **hauteur** d'un arbre est la <u>plus grande profondeur</u> d'une feuille de l'arbre.

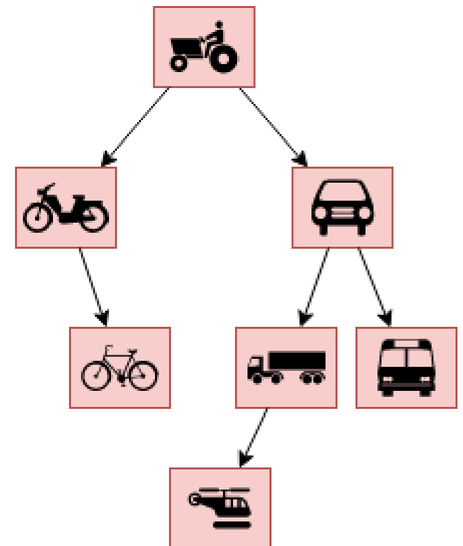
La **taille** d'un arbre est son <u>nombre de nœuds</u>.



Cette structure de donnée est **récursive** : chaque *nœud* est lui-même *nœud-racine* d'un sous-arbre (également appelé **branche**)

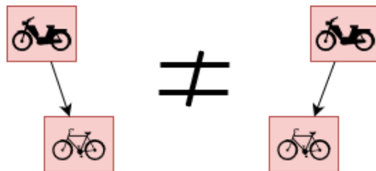
<u>Remarque importante :</u> il n'existe pas de définition universelle pour la hauteur d'un arbre et la profondeur d'un nœud dans un arbre.
Dans certains cas la profondeur des nœuds est comptée à partir de 1 et/ou la hauteur est égale au nombre de profondeurs différentes …
Parfois également, la taille d'un arbre ne tient pas compte des feuilles !!

2- CAS PARTICULIER DES ARBRES BINAIRES :



Les **arbres binaires** (**AB**) sont des cas particuliers d'arbres où chaque *nœud* possède <u>au maximum **deux** ***fils***</u>.

Attention, les fils d'un nœud sont <u>classés</u> : un fils droit et/ou un fils gauche. Ils ne sont pas intervertibles !



3- DIFFERENTS TYPES D'ARBRES BINAIRES :

Il est possible d'avoir des arbres binaires de même taille mais de « forme » très différente :

Arbre **parfait** : tous ses nœuds possèdent <u>exactement 2 fils</u> (sauf les feuilles qui en ont zéro !)	Arbre **(presque) complet**: toutes ses feuilles sont à la même profondeur et les feuilles manquantes sont toutes à droite	Arbre **équilibré** : toutes ses feuilles sont à la même profondeur	Arbre **filiforme** (ou **dégénéré**) : tous ses nœuds possèdent <u>un unique fils</u> (on parle aussi de **peigne**)

On note n la taille d'un arbre et h sa hauteur

On considère les formes d'arbre les plus « extrêmes » et la relation entre leur taille et leur hauteur est :

- arbre filiforme : $n = h + 1$
  ou $n = h$ si on considère qu'un arbre vide a une hauteur nulle
- arbre parfait : $n = 2^{h+1} - 1$

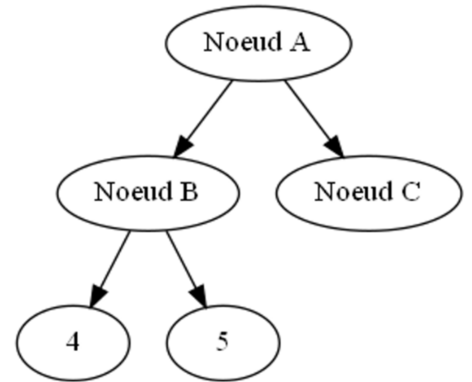
On peut encadrer la taille n d'un arbre binaire que l'on peut obtenir pour une hauteur h donnée :

$$h + 1 \leq n \leq 2^{h+1} - 1$$

<u>Exemple</u> : la taille d'un ardre de hauteur 3 est comprise entre

4- VISUALISATION DES GRAPHES AVEC « GRAPHVIZ » :

Pour visualiser nos arbres, on utilisera le logiciel *Graphviz* qui un logiciel open source de visualisation de graphes, développé en langage C par la société américaine AT&T. Pour l'installer, il ne suffit pas d'importer la librairie graphviz à partir de la console python (*pip install graphviz*), mais il faut **aussi** installer le logiciel sur le disque dur (liens d'installations sur le site https://graphviz.org/download/).



Le package installé propose 2 classes nommées *Graph* et *Digraph*. La première classe permet de tracer des graphes non orientés, la seconde, des graphes orientés. Pour le tracé d'arbres, on utilisera *Digraph*.

Le code pour tracer l'arbre donné ci-contre sera le suivant :

```
from graphviz import Digraph
graphe = Digraph(format = 'png' , filename = 'arbre')

graphe.node('1', 'Noeud A')
graphe.node('2' ,'Noeud B')
graphe.node('3' ,'Noeud C')
graphe.node('4')
graphe.node('5')

graphe.edge('1','2')
graphe.edge('1','3')
graphe.edge('2','4')
graphe.edge('2','5')

graphe.view()
```

Création de l'objet graphe qui est une instance de la classe *Digraph*

Importation de la classe

Nom du fichier qui sera ici arbre.png

Le graphe orienté généré sera une image au format png (autre possibilité au format *pdf*)

En appliquant la méthode *node()* à l'objet graphe, on crée un nœud, ici d'identifiant '3'. L'identifiant doit être un *string*. Le label, ici 'Nœud C' est facultatif. C'est cette valeur qui sera affichée.

En appliquant la méthode *view()* à l'objet graphe on crée ici le fichier arbre.png

En appliquant la méthode *edge()* à l'objet graphe, on crée une arête orientée, ici entre les nœuds d'identifiant '2' et '5'

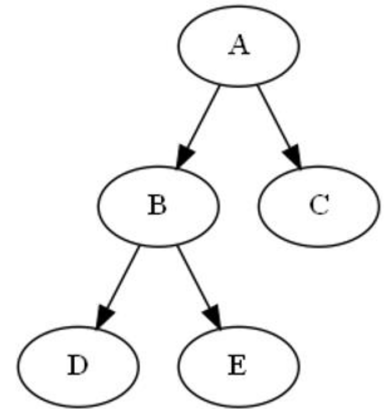
Remarque :

Il est possible de tracer le même arbre, sans définir au préalable chacun des nœuds. On applique alors uniquement la méthode *edge()* à l'objet graphe. On met en argument les labels qui seront les valeurs affichées dans l'arbre.

```
graphe.edge("Noeud A","Noeud B")
graphe.edge("Noeud A","Noeud C")
graphe.edge("Noeud B","4")
graphe.edge("Noeud B","5")

graphe.view()
```

5- DIFFERENTES POSSIBILITES D'IMPLEMENTER UN ARBRE BINAIRE :



Il existe différentes possibilités d'implémenter un arbre. On peut utiliser des listes, des dictionnaires ou une classe en POO. On voit cela pour implémenter l'arbre donné ci-contre :

a. IMPLEMENTATION AVEC DES LISTES :

Pour traiter cet arbre, on pourrait utiliser la liste de listes suivante :

```
arbre = ['A', ['B',['D',None,None],['E',None,None]] , ['C',None,None] ]
```

b. IMPLEMENTATION AVEC DES DICTIONNAIRES :

Pour implémenter cet arbre, on pourrait utiliser les dictionnaires suivants :

```
arbre = {'info' : 'A' ,
         'fg' : {'info' : 'B' ,
                 'fg' : {'info':'D' , 'fg':None ,'fd':None}
                 'fd' : {'info':'E' , 'fg':None ,'fd':None}
                }
         'fd' : {'info':'C' , 'fg':None ,'fd':None},
        }
```

c. IMPLEMENTATION AVEC UNE CLASSE :

Pour implémenter un arbre on peut utiliser la POO et créer une classe comme celle définie ci-dessous :

```
class Arbre :
    def __init__(self, info = None , fg = None , fd = None) :
        self.info = str(info)
        self.fg = fg
        self.fd = fd
```

Pour traiter l'arbre qui nous intéresse ici, il serait nécessaire d'exécuter :

```
D = Arbre('D')
E = Arbre('E')
B = Arbre('B',D,E)
C = Arbre('C')
A = Arbre('A',B,C)
```

Ou

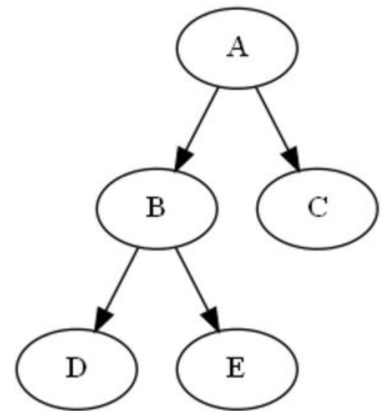
```
A = Arbre('A',
          Arbre('B',
                Arbre('D') ,
                Arbre('E')
               ) ,
          Arbre('C')
          )
```

Ou encore :

```
A = Arbre('A', Arbre('B', Arbre('D') , Arbre('E')) , Arbre('C'))
```

6- COMMENT PARCOURIR UN ARBRE BINAIRE :

Quelle que soit l'implémentation utilisée, on constate qu'un arbre est composé d'un nœud racine *(nd)* qui contient l'information que l'on cherche à stocker et de 2 sous-arbres que l'on nommera « *fils gauche (fg)*» et « *fils droit (fd)*». Chacun des sous-arbres est lui-même un arbre à part entière.
On obtient ainsi un ensemble dans lequel plusieurs arbres élémentaires *(nd + fg + fd)* sont imbriqués les uns dans les autres. Cette architecture est particulièrement adaptée aux algorithmes de type **récursifs**. On présente dans la suite de tels algorithmes pour réaliser un parcours d'arbre. Une version itérative est également donnée.



Ces algorithmes sont présentés dans un contexte d'implémentation POO qui utilise une classe Arbre.

a. METHODE RECURSIVE DITE « PREFIXE » :

```
def parcoursRecursif(self) :
    print(self.info)
    if self.fg != None : self.fg.parcoursRecursif()
    if self.fd != None : self.fd.parcoursRecursif()
```

Avec :

```
A = Arbre('A', Arbre('B', Arbre('D') , Arbre('E')) , Arbre('C'))
```

En exécutant `>>> A.parcoursRecursif()` il se produit les appels suivants (notation ***parRec()***)

Affichage dans la console :

b. METHODE RECURSIVE DITE « INFIXE » :

```
def parcoursRecursif(self) :
    if self.fg != None : self.fg.parcoursRecursif()
    print(self.info)
    if self.fd != None : self.fd.parcoursRecursif()
```

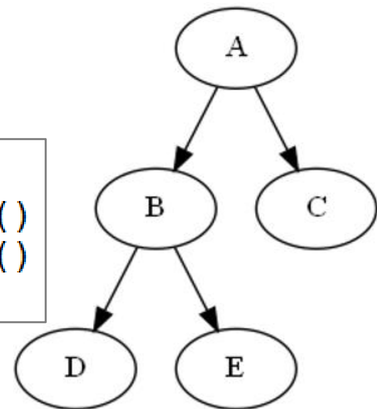
Avec :

```
A = Arbre('A', Arbre('B', Arbre('D') , Arbre('E')) , Arbre('C'))
```

En exécutant `>>> A.parcoursRecursif()` , il se produit les appels suivants :

Affichage dans la console :

c. METHODE RECURSIVE DITE « POSTFIXE » :




```
def parcoursRecursif(self) :
    if self.fg != None : self.fg.parcoursRecursif()
    if self.fd != None : self.fd.parcoursRecursif()
    print(self.info)
```

Avec :

```
A = Arbre('A', Arbre('B', Arbre('D') , Arbre('E')) , Arbre('C'))
```

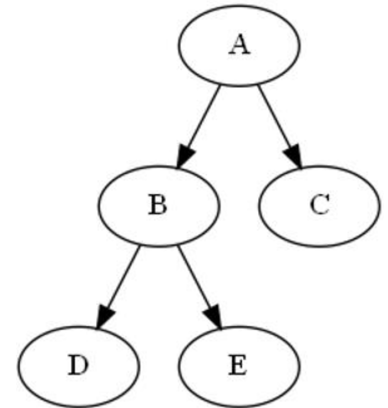
En exécutant `>>> A.parcoursRecursif()` , il se produit les appels suivants :

Affichage dans la console :

d. METHODE ITERATIVE DITE « PARCOURS EN LARGEUR » :

En utilisant une structure de **file**, le parcours itératif est extrêmement simple.

```
def parcoursLargeur(self) :
    f = File()
    f.enfiler(self)
    while not f.estVide() :
        nd = f.defiler()
        print(nd.info)
        if nd.fg != None : f.enfiler(nd.fg)
        if nd.fd != None : f.enfiler(nd.fd)
```



Avec :

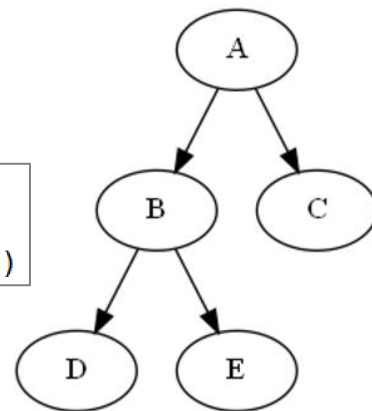
```
A = Arbre('A', Arbre('B', Arbre('D') , Arbre('E')) , Arbre('C'))
```

En exécutant `>>> A.parcoursLargeur()` , il se produit les exécutions suivantes :

Etat de la file f	*Exécution*

7- CALCUL DE LA TAILLE D'UN ARBRE :

Pour calculer la taille d'un arbre, c'est-à-dire le nombre de nœuds d'un arbre, le code suivant pourrait-il convenir ?




```
def taille(self) :
    if self.fg != None and self.fd != None :
        return 1 + self.fg.taille() + self.fd.taille()
```

Quelles sont les choses qui ne vont pas ?

Pour corriger les problèmes mis en évidence, on peut proposer le code suivant :

```
def taille(self) :
    if self.fg == None and self.fd == None :
        return 1
    if self.fg == None and self.fd != None :
        return 1 + self.fd.taille()
    if self.fg != None and self.fd == None :
        return 1 + self.fg.taille()
    if self.fg != None and self.fd != None :
        return 1 + self.fg.taille() + self.fd.taille()
```

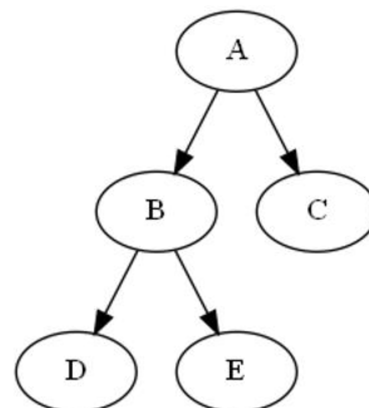
Ou la version suivante qui est légèrement plus épurée :

```
def taille(self) :
    if self.fg == None and self.fd == None :
        return  1
    elif self.fg == None : # seul self.fd existe
        return 1 + self.fd.taille()
    elif self.fd == None :# seul self.fg existe
        return 1 + self.fg.taille()
    else : # self.fg et self.fd existent
        return 1 + self.fg.taille() + self.fd.taille()
```

Une version encore plus épurée de ce même code est proposée ci-dessous :

```
def taille(self) :
    if self.fg == None : nbG = 0
    else : nbG = self.fg.taille()
    if self.fd == None : nbD = 0
    else : nbD = self.fd.taille()
    return 1 + nbG + nbD
```

Avec :



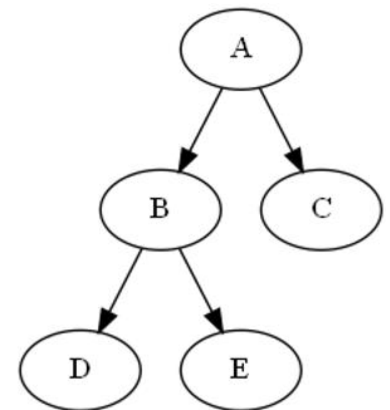

```
A = Arbre('A', Arbre('B', Arbre('D') , Arbre('E')) , Arbre('C'))
```

En exécutant `>>> A.taille()` , il se produit les exécutions suivantes :

8- CALCUL DE LA HAUTEUR D'UN ARBRE :

Pour calculer la hauteur d'un arbre, un algorithme récursif peut être le suivant :

```
def hauteur(self) :
    if self.fg == None : hG = -1
    else : hG = self.fg.hauteur()
    if self.fd == None : hD = -1
    else : hD = self.fd.hauteur()
    return 1 + max(hG , hD)
```



Avec :

```
A = Arbre('A', Arbre('B', Arbre('D') , Arbre('E')) , Arbre('C'))
```

En exécutant `>>> A.hauteur()` , il se produit les exécutions suivantes :