

# Algorithmique et structures de données

## Exercices INF TC1

Romain Vuillemot (Math-Info) — École Centrale de Lyon

2021-2022

## Structures de données

### Transformation

**Exercice.** Comment passer d'une structure de données à l'autre ?

```
{'a': 1, 'b': 2, 'er': 1, 'gh': 2}
```

vers

```
(['a', 'b', 'er', 'gh'], [1, 2, 1, 2])
```

**Exercice.** Comment passer d'une structure de données à l'autre ?

```
{'a': 1, 'b': 2, 'er': 1, 'gh': 2}
```

vers

```
(['a', 'b', 'er', 'gh'], [1, 2, 1, 2])
```

**Solution.**

```
hist = {'a': 1, 'b': 2, 'er': 1, 'gh': 2}
cles = []
vals = []
for k, v in hist.items():
    cles.append(k)
    vals.append(v)
cles, vals
```

Ou bien plus simplement

```
list(a.keys()), list(a.values())
```

**Exercice.** Que réalise l'algorithme suivant ? (La classe Noeud représente un arbre)

```
class Noeud:
    ... def __init__(self, v, c = []):
    ...     self.v = v
    ...     self.c = c

def parcours(n):
    ... stack = []
    ... stack.append(n)
    ... r = []
    ... while len(stack) > 0:
    ...     current = stack.pop(0)
    ...     r.append(current.v)
    ...     for v in current.c:
    ...         stack.append(v)
    ... return r
```

**Exercice.** Que réalise l'algorithme suivant ? (La classe Noeud représente un arbre)

```
class Noeud:
    ... def __init__(self, v, c = []):
    ...     self.v = v
    ...     self.c = c

    def parcours(n):
        ... stack = []
        ... stack.append(n)
        ... r = []
        ... while len(stack) > 0:
        ...     current = stack.pop(0)
        ...     r.append(current.v)
        ...     for v in current.c:
        ...         stack.append(v)
        ... return r
```

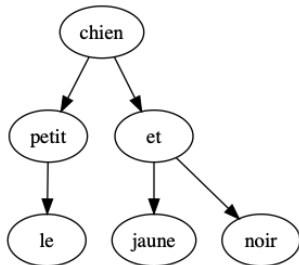
**Solution.**

- ▶ Parcours en largeur (de l'arbre)
- ▶ Renvoie la liste des noeuds parcourus

## Arbres

### Propriétés d'un arbre

**Exercice.** Ci-dessous un arbre (binaire). Proposez un parcours afin d'obtenir les noeuds dans un ordre cohérent (*le petit chien jaune..*).



```
from graphviz import Digraph

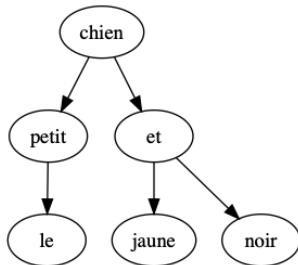
name = 'arbre—viz—profondeur'

g = Digraph('G', filename = name + '.gv', \
            format='png') # pdf par défaut

g.edge('chien', 'petit')
g.edge('chien', 'et')
g.edge('petit', 'le')
g.edge('et', 'jaune')
g.edge('et', 'noir')

# génère et affiche le graphe
g.view()
```

**Exercice.** Ci-dessous un arbre (binaire). Proposez un parcours afin d'obtenir les noeuds dans un ordre cohérent (*le petit chien jaune..*).



```
from graphviz import Digraph

name = 'arbre-viz-profondeur'

g = Digraph('G', filename = name + '.gv', \
            format='png') # pdf par défaut

g.edge('chien', 'petit')
g.edge('chien', 'et')
g.edge('petit', 'le')
g.edge('et', 'jaune')
g.edge('et', 'noir')

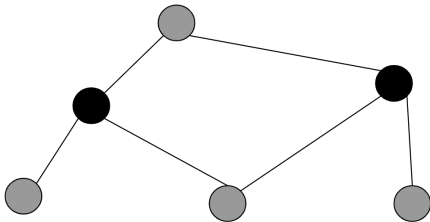
# génère et affiche le graphe
g.view()
```

**Solution.**

- ▶ Structure de donnée d'arbre en Objet ou en Dictionnaire
- ▶ Parcours en profondeur *infixe* (G R D)

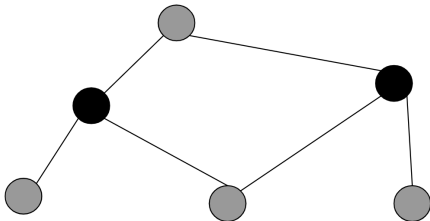
<https://gitlab.ec-lyon.fr/rvuillemin/inf-tci/-/blob/master/TD01/code/arbre-parcours-profondeur.py>

**Exercice.** Etant donné le graphe ci-dessous, est-il 2-colorié ? (est-ce que deux noeuds voisins n'ont pas la même couleur ?)





**Exercice.** Etant donné le graphe ci-dessous, est-il 2-colorié ? (est-ce que deux noeuds voisins n'ont pas la même couleur ?)



### Solution.

- ▶ Une structure de données de graphe qui stocke l'information de couleur et parcours en largeur (par exemple)
- ▶ Vérification de la couleur entre noeud courant et voisins
- ▶ Pour la 2-coloration voir le cours sur les algorithmes Gloutons

<https://gitlab.ec-lyon.fr/rvuilleminf-tcl/-/blob/master/TD01/code/graph-check-color.py>

**Exercice.** Comment sortir de ce labyrinthe ?

(l'entrée est le coin en haut à gauche et la sortie le coin en bas à droite, les 1 sont des murs)

```
labyrinthe = [[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
               [0, 1, 0, 1, 1, 1, 1, 1, 1, 0],  
               [0, 1, 0, 1, 0, 0, 0, 0, 1, 0],  
               [0, 1, 0, 1, 0, 1, 1, 0, 1, 0],  
               [0, 1, 0, 1, 0, 1, 1, 0, 1, 0],  
               [0, 1, 0, 1, 0, 0, 1, 0, 1, 0],  
               [0, 1, 0, 1, 1, 1, 1, 0, 1, 0],  
               [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],  
               [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],  
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

**Exercice.** Comment sortir de ce labyrinthe ?

(l'entrée est le coin en haut à gauche et la sortie le coin en bas à droite, les 1 sont des murs)

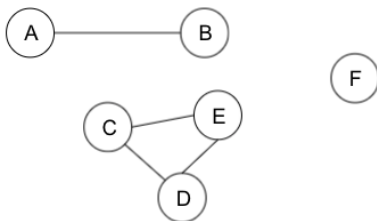
```
labyrinthe = [[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
               [0, 1, 0, 1, 1, 1, 1, 1, 1, 0],  
               [0, 1, 0, 1, 0, 0, 0, 0, 1, 0],  
               [0, 1, 0, 1, 0, 1, 1, 0, 1, 0],  
               [0, 1, 0, 1, 0, 1, 1, 0, 1, 0],  
               [0, 1, 0, 1, 0, 0, 1, 0, 1, 0],  
               [0, 1, 0, 1, 1, 1, 1, 0, 1, 0],  
               [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],  
               [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],  
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

**Solution.**

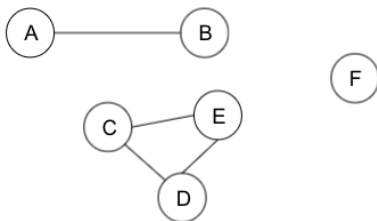
- ▶ On considère la matrice comme un graphe avec une relation de voisinage entre cases
- ▶ On peut effectuer un parcours en profondeur avec mémorisation du chemin  
parcours sous forme de liste

<https://gitlab.ec-lyon.fr/rvuillemin/inf-tci/-/blob/master/TD01/code/graph-labyrinthe.py>

**Exercice.** Identifiez les composantes connexes de ce graphe



**Exercice.** Identifiez les composantes connexes de ce graphe

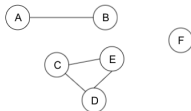


**Solution.**

- ▶ Structure de données d'arbre  $G = "A": ["B"], \dots$
- ▶ Algorithme de parcours en profondeur + marquage de sommets visités
- ▶ Pour tous les sommets (si plusieurs groupes de sommets visités alors plusieurs composantes connexes)

<https://gitlab.ec-lyon.fr/rvuilleminf-tcl/-/blob/master/TD01/code/graph-labyrinthe.py>

**Exercice.** Identifiez les composantes connexes de ce graphe



```

def recherche_composantes_connexes(graph):
    visite = []
    composantes_connexes = []
    for n in graph.keys():
        if n not in visite:
            cc = []
            visite, cc = parcours_profondeur(graph, n, visite, cc)
            composantes_connexes.append(cc)
    return composantes_connexes

def parcours_profondeur(graph, start, visite, chemin):
    if start in visite:
        return visite, chemin
    visite.append(start)
    chemin.append(start)
    for n in graph[start]:
        visite, chemin = parcours_profondeur(graph, n, visite, chemin)
    return visite, chemin

G = {"A": ["B"], "B": ["A"], "C": ["D", "E"], "D": ["C", "E"], "E": ["C", "D"], "F": []}

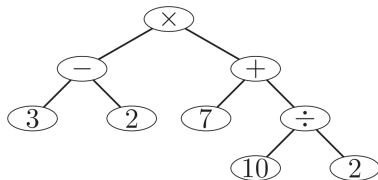
composantes_connexes = recherche_composantes_connexes(G)

print("Nombre de composantes connexes =", len(composantes_connexes))

# affichage de chacune des composantes
for cc in composantes_connexes:
    print(cc)
  
```

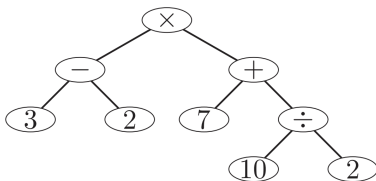
## Validation d'arbre syntaxique

**Exercice.** Nous souhaitons évaluer l'expression arithmétique suivante  $(3 - 2) * (7 + 10/2)$  stockée sous la forme d'un arbre comme ci-dessous. Quelle est la structure de données d'arbre Python et le mécanisme de parcours permettant le calcul de l'expression ?



## Validation d'arbre syntaxique

**Exercice.** Nous souhaitons évaluer l'expression arithmétique suivante  $(3 - 2) * (7 + 10/2)$  stockée sous la forme d'un arbre comme ci-dessous. Quelle est la structure de données d'arbre Python et le mécanisme de parcours permettant le calcul de l'expression ?

**Solution.**

- ▶ Structure de données d'arbre binaire en POO



<https://gitlab.ec-lyon.fr/rvuillemin/inf-tc1/-/blob/master/TD01/code/arbre-syntaxique.py>



## Exercice. Validation d'arbre syntaxique



```

class Noeud:
    def __init__(self, v=None, g=None, d=None):
        self.valeur = v
        self.gauche = g
        self.droit = d

def valide(r):
    if r == None:
        return True
    elif r.valeur in ['+', '-', '*', '/']:
        return valide(r.gauche) and valide(r.droit)
    elif r.gauche == None and r.droit == None:
        if not isinstance(r.valeur, int):
            return False
        else:
            return True
    else:
        return False

# (3-2) * (7 + 10 / 2)
arbre = Noeud('*', Noeud('-', Noeud(3), Noeud(2)), Noeud('+', Noeud(7),
    Noeud('/', Noeud(10), Noeud(2))))

print(valide(arbre))

def eval(r):
    if valide(arbre) is not True:
        return None

    if r.valeur == '+':
        return eval(r.gauche) + eval(r.droit)
    elif r.valeur == '*':
        return eval(r.gauche) * eval(r.droit)
    elif r.valeur == '-':
        return eval(r.gauche) - eval(r.droit)
    elif r.valeur == '/':
        return eval(r.gauche) / eval(r.droit)
  
```