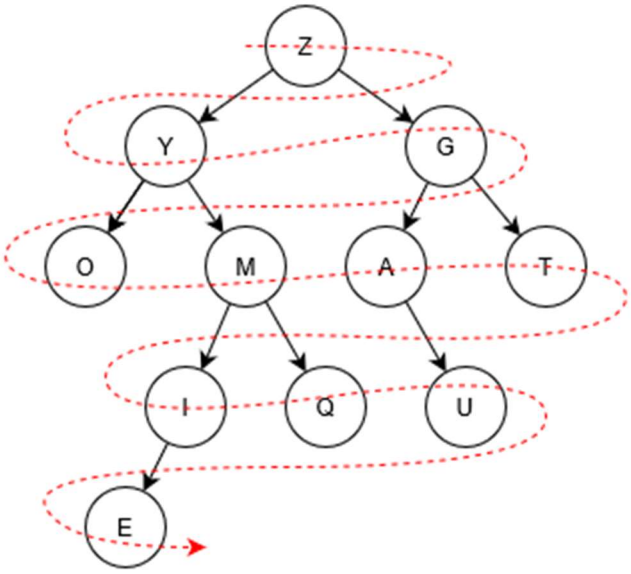
Introduction :

- L'**algorithme de parcours en largeur** (ou BFS, pour *Breadth-First Search* en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc.
- L'**algorithme de parcours en profondeur** (ou **DFS**, pour *Depth-First Search* en anglais), l'exploration s'effectue depuis un sommet S, comme suit : il poursuit alors un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité.

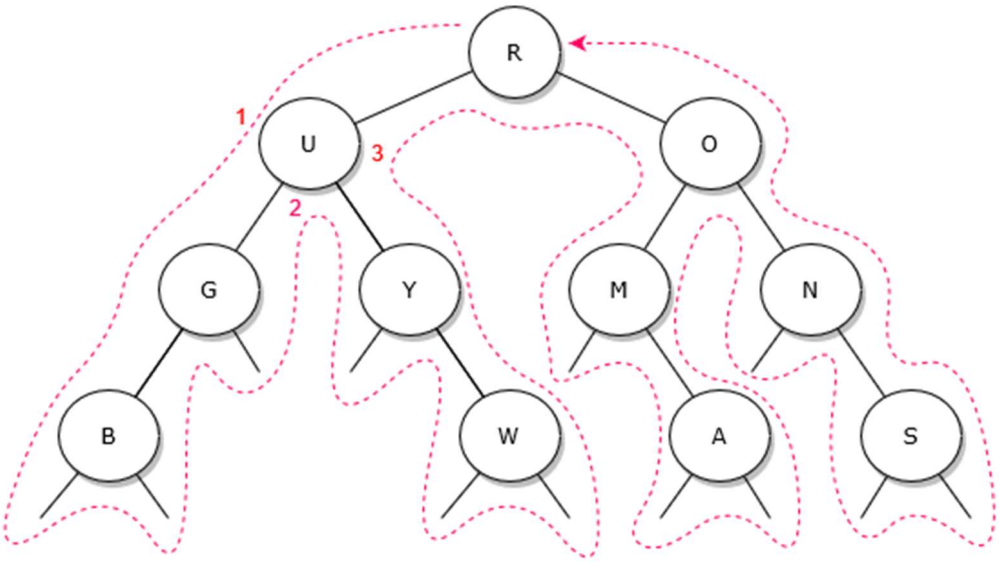
Représentation :

On se balade autour des arbres en suivant les pointillés.

- **Parcours en largeur :**



- **Parcours en profondeur :**



Dans le schéma ci-dessus, on a rajouté des "nœuds fantômes" pour montrer que l'on peut considérer que chaque nœud est visité 3 fois :

- Une fois par la gauche.
- Une fois par en dessous.
- Une fois par la droite

Parcours en largeur (programmation) :

- ***On utilise une file.***
- ***On met l'arbre dans la file.***
- ***Puis tant que la file n'est pas vide :***
  - ***On défile la file.***
  - ***On récupère sa racine.***
    - ***On enfile son fils gauche s'il existe.***
    - ***On enfile son fils droit s'il existe.***

J'ai fait le choix de représenter graphiquement notre arbre, nous allons donc avoir besoin de deux modules externes : Matplotlib et Networkx.
On les installe donc sur notre OS, en tapent les commandes suivant dans le terminal :

```
pip install matplotlib
pip install networkx
```

1. Importation des modules externes dans le code :

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
```

2. Création de l'arbre :

```
 4 # Insertion des noeuds :
 5 def Noeud(nom, fg = None, fd = None) :
 6     return {'racine': nom, 'fg' : fg, 'fd': fd}
 7
 8 # Définition des noeuds :
 9 Q = Noeud('Q')
10 P = Noeud('P')
11 M = Noeud('M', P, Q)
12 L = Noeud('L')
13 G = Noeud('G', L, M)
14 K = Noeud('K')
15 J = Noeud('J')
16 F = Noeud('F', J, K)
17 C = Noeud('C', F, G)
18 O = Noeud('O')
19 N = Noeud('N')
20 I = Noeud('I', N, O)
21 H = Noeud('H')
22 E = Noeud('E', H, I)
23 D = Noeud('D')
24 B = Noeud('B', D, E)
```

```
racine = Noeud('A', B, C)

# Création de l'arbre :
def Construit(arbre) :
    if arbre == None:
        return None
    else:
        return(arbre['racine'],Construit(arbre['fg']),Construit(arbre['fd']))
```

3. Calcul de la hauteur de l'arbre :

```
# Calcul de la hauteur de l'arbre :
def hauteur(arbre):
    if arbre == None :
        return -1
    else :
        h1 = 1 + hauteur(arbre[1])
        h2 = 1 + hauteur(arbre[2])
        return max(h1,h2)
```

4. Représentation graphique de l'arbre étudié :

Pour cette partie je me suis fortement inspiré du programme écrit par Van Zuijlen Stéphan.
Je l'ai modifié pour qu'il s'adapte parfaitement à mon utilisation !

```
# Représentation graphique de l'arbre binaire :
def repr_graph(arbre, size=(8,8), null_node=False):
    """
    size : tuple de 2 entiers. Si size est int -> (size, size)
    null_node : si True, trace les liaisons vers les sous-arbres vides
    """
    def parkour(arbre, noeuds, branches, labels, positions, profondeur,
pos_courante, pos_parent, null_node):
        if arbre != None:
            noeuds[0].append(pos_courante)
            positions[pos_courante] = (pos_courante, profondeur)
            profondeur -= 1
            labels[pos_courante] = str(arbre[0])
            branches[0].append((pos_courante, pos_parent))
            pos_gauche = pos_courante - 2**profondeur
            parkour(arbre[1], noeuds, branches, labels, positions,
profondeur, pos_gauche, pos_courante, null_node)
            pos_droit = pos_courante + 2**profondeur
            parkour(arbre[2], noeuds, branches, labels, positions,
profondeur, pos_droit, pos_courante, null_node)
        elif null_node:
            noeuds[1].append(pos_courante)
            positions[pos_courante] = (pos_courante, profondeur)
            branches[1].append((pos_courante, pos_parent))


    if arbre == None :
        return

    branches = [[]]
    profondeur = hauteur(arbre)
    pos_courante = 2**profondeur
    noeuds = [[pos_courante]]
```

```
 73     positions = {pos_courante: (pos_courante, profondeur)}
 74     labels = {pos_courante: str(arbre[0])}
 75
 76     if null_node:
 77         branches.append([])
 78         noeuds.append([])
 79
 80     profondeur -= 1
 81     parkour(arbre[1], noeuds, branches, labels, positions, profondeur,
pos_courante - 2**profondeur, pos_courante, null_node)
 82     parkour(arbre[2], noeuds, branches, labels, positions, profondeur,
pos_courante + 2**profondeur, pos_courante, null_node)
 83
 84     mon_arbre = nx.Graph()
 85
 86     if type(size) == int:
 87         size = (size, size)
 88     plt.figure(figsize=size)
 89
 90     nx.draw_networkx_nodes(mon_arbre, positions, nodelist=noeuds[0],
node_color="white", node_size=550, edgecolors="blue", margins=0.1)
 91     nx.draw_networkx_edges(mon_arbre, positions, edgelist=branches[0],
edge_color="black", width=2)
 92     nx.draw_networkx_labels(mon_arbre, positions, labels)
 93
 94     if null_node:
 95         nx.draw_networkx_nodes(mon_arbre, positions, nodelist=noeuds[1],
node_color="white", node_size=50, edgecolors="grey")
 96         nx.draw_networkx_edges(mon_arbre, positions,
edgelist=branches[1], edge_color="grey", width=1)
 97
 98     ax = plt.gca()
 99     ax.margins(0.1)
100     plt.axis("off")
101     plt.show()
102     plt.close()
```

5. Parcours en Largeur :

```
104 # Parcours Largeur
105
106 def ParcoursLargeur(arbreB):
107     file = []           # On génère la file vide.
108     file.append(arbreB) # On met l'arbre dans la file.
109     res = []            # On génère une liste pour le résultat final.
110
111     while len(file) > 0:         # Tant que la file n'est pas vide.
112       node = file.pop(0)         # On défile la file.
113       res.append(node["racine"]) # On récupère la racine du nœud étudié.
114
115       if node["fg"] != None :
116         file.append(node["fg"])  # On enfile son fils gauche s'il existe.
117
118       if node["fd"] != None :
119         file.append(node["fd"])  # On enfile son fils droit s'il existe.
120
121     return res          # On retourne le résultat.
```

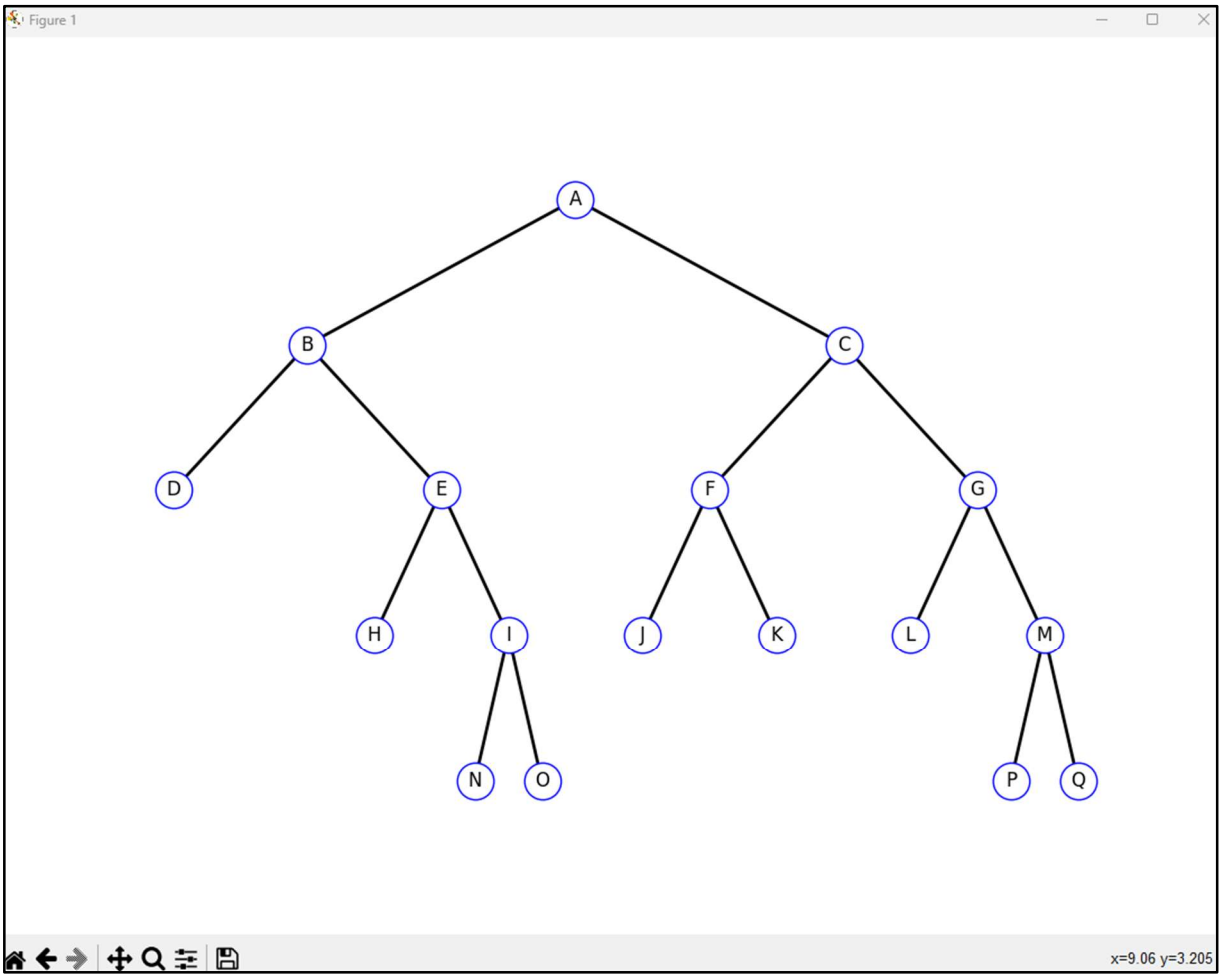
6. Appel des fonctions :

```
129 print("\nParcours en Largeur :")
130
131 # On affiche chaque nœud via le parcours en largeur :
132 # (en séparant chaque nœud par un tiret)
133 for node_larg in ParcoursLargeur(racine) :
134   print(node_larg, end='')
135   if node_larg != ParcoursLargeur(racine)[-1]:
136     print(' - ', end='')
137
138 repr_graph(arbre,(11,8),False)  # Représentation graphique
```

7. Résultats :

```
('A', ('B', ('D', None, None), ('E', ('H', None, None), ('I', ('N',
None, None), ('O', None, None)))), ('C', ('F', ('J', None, None),
('K', None, None)), ('G', ('L', None, None), ('M', ('P', None, None),
('Q', None, None)))))
Hauteur de l'arbre : 4

Parcours en Largeur :
A - B - C - D - E - F - G - H - I - J - K - L - M - N - O - P - Q
```



Parcours en profondeur (programmation) :

Pour le fonctionnement des parcours en profondeur, se référer au schéma en début de document.

On utilisera les mêmes fonctions que celles précédemment utilisées pour la création de l'arbre. De même on utilisera le même arbre binaire, dont vous pouvez d'ailleurs voir la représentation sur la page précédente.
Nous allons voir qu'il existe trois types de parcours en profondeur, nous allons donc les implémenter dans notre code !

1. Parcours « préfixe » :

Dans un parcours « préfixe », on liste le nœud la première fois qu'on le rencontre.

```
1 def Parcours_Prefixe(arbreB):
2     if arbreB != None :                 # Si l'arbre n'est pas vide :
3         print(arbreB[0],'- ',end='')    # J'affiche le nœud courant
(affichage sur une même ligne)
4         Parcours_Prefixe(arbreB[1])     # Je rappelle récursivement ma
fonction (sous-arbre gauche)
5         Parcours_Prefixe(arbreB[2])     # Je rappelle récursivement ma
fonction (sous-arbre droit)
```

Résultat :

```
1 Parcours Prefixe :
2 A - B - D - E - H - I - N - O - C - F - J - K - G - L - M - P - Q
```

2. Parcours « infixe » :

Dans un parcours « infixe », on liste le nœud la seconde fois qu'on le rencontre.

```
1 def Parcours_Infixe(arbreB):
2     if arbreB != None :                 # Si l'arbre n'est pas vide :
3         Parcours_Infixe(arbreB[1])      # Je rappelle récursivement ma
fonction (sous-arbre gauche)
4         print(arbreB[0],'- ',end='')    # J'affiche le nœud courant
5         Parcours_Infixe(arbreB[2])      # Je rappelle récursivement ma
fonction (sous-arbre droit)
```

Résultat :

```
1 Parcours Infixe :
2 D - B - H - E - N - I - O - A - J - F - K - C - L - G - P - M - Q
```

3. Parcours « suffixe » :

Dans un parcours « suffixe », on liste le nœud la dernière fois qu'on le rencontre.

```
1 def Parcours_Suffixe(arbreB):
2     if arbreB != None :                 # Si l'arbre n'est pas vide :
3         Parcours_Suffixe(arbreB[1])     # Je rappelle récursivement ma
fonction (sous-arbre gauche)
4         Parcours_Suffixe(arbreB[2])     # Je rappelle récursivement ma
fonction (sous-arbre droit)
5         print(arbreB[0],'- ',end='')    # J'affiche le nœud courant
```

Résultat :

```
1 Parcours Suffixe :
2 D - H - N - O - I - E - B - J - K - F - L - P - Q - M - G - C - A
```

Pour une meilleure compréhension du programme, je vous fournis ci-joint le code source Python afin que vous puissiez le tester sur votre propre machine ! Vous pouvez aussi modifier vous-même l'arbre de test, et notamment les nœuds le composant !