DJIKOLOUM ARMAND & TOMTE HASSANE

**PARCOURS D’UN ARBRE**

**Introduction**

Pour écrire des algorithmes effectuant des opérations sur des informations stockées en structure de donnée non linéaire : arbre. On a besoin de parcourir tous les nœuds de l’arbre **(parcours intégral)** ou bien tous les nœuds jusqu’à ce que l’on trouve ce que l’on cherche **(parcours de recherche**). Un parcours d'arbre est une façon d'ordonner les nœuds d'un arbre afin de les parcourir. On peut le voir comme une fonction qui a un arbre associe une liste de ses nœuds, même si la liste n'est souvent pas explicitement construite par le parcours.

Nous allons tout long de ce tutoriel, parler des arbres binaires, définir les terminologies très liées au parcours d’arbre, et porter une attention plus particulière aux différents types de parcours d’arbre, sur lesquels nous étudierons leurs stratégies et leurs implémentations algorithmiques. Nous terminerons avec des exemples d’applications.

**Objectif Général : C**

Comprendre puis appliquer les algorithmes de parcourt des arbres.

**Objectifs Spécifiques :**

* Connaitre les parcours en profondeur : préfixe, infixe et postfixe d’un arbre binaire.
* Connaitre le parcours en largeur d’un arbre binaire.
* Savoir coder les parcours d’un arbre binaire en Python.

**Définitions**

**Arbre :** Un arbre est une structure de donnée non linéaire, c’est-à-dire une manière particulière de représenter, de structurer l’information dans la mémoire de l’ordinateur.

**Racine :** est l’élément de base d’un arbre : De ce nœud racine peuvent ensuite partir des arêtes reliant d’autres nœuds, lesquels peuvent eux-mêmes relier d’autres nœuds pour former des branches.

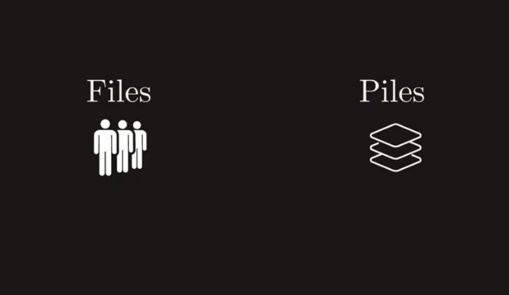
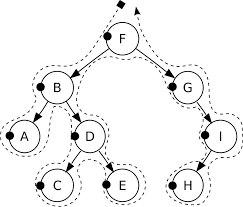
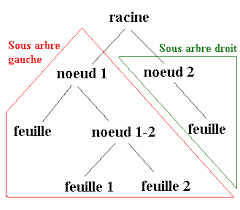
Un **sous-arbre** : est une portion d’arbre à partir d’un nœud quelconque qui constitue la racine de ce sous-arbre.

**Parcours d’arbre :** examiner les nœuds d’un arbre pour effectuer un traitement.

**Files :** Les files sont des structures de données linéaires basées sur le principe FIFO (First In First Out : le premier qui est rentré sera le premier à sortir.

**Piles : Les** piles sont basées sur le principe LIFO (Last In First Out : le dernier rentré sera le premier à sortir). On retrouve souvent ce principe LIFO en informatique.

**Figure 1:illustrations des définitions**



**Types de parcours classiques : deux directions**

**Parcours en largeur : Breadth First Search (BFS)**

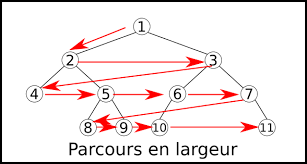
Définition : Le parcours en largeur ou itératif noté BFS consiste à parcourir l'arbre niveau par niveau. Les nœuds de niveau 0 sont d'abord parcourus puis les nœuds de niveau 1 et ainsi de suite.

Dans chaque niveau, les nœuds sont parcourus de la gauche vers la droite.

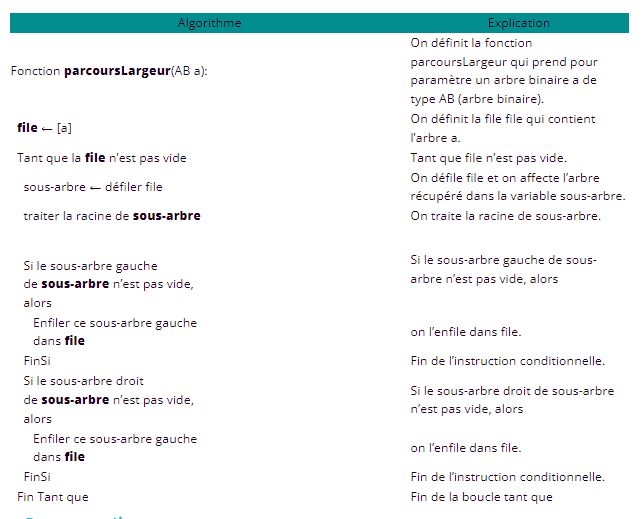
**Stratégie**

Le parcours en largeur se programme à l'aide d'une file (FIF0) de la manière suivante :

1. Créer une file de nœuds à traiter, au début que la racine ;
2. Tant que la file n’est pas vide, pour le premier nœud :

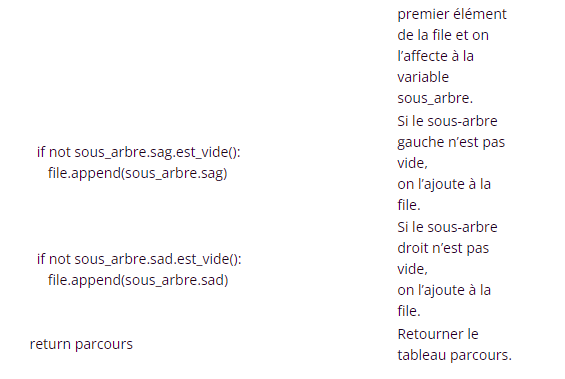
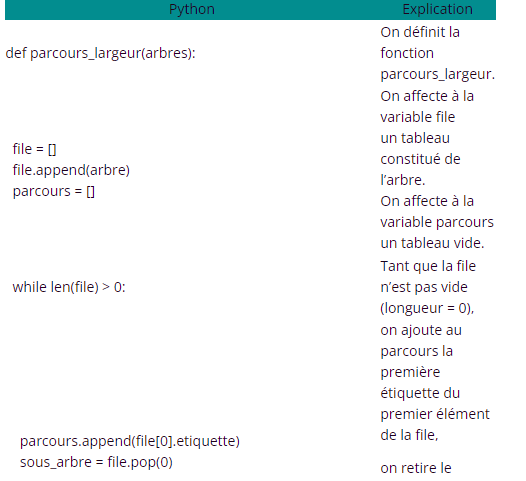
* Ajouter ses enfants à la file ;
* Traiter le nœud ;
* Retirer de la file.

L'arbre ci-dessus parcours les nœuds dans l'ordre **[1,2,3,4,5,6,7,8,9,10,11].**

Algorithme

**Programmation**

En Python, la fonction parcourslargeur(arbre) implémente le parcours en largeur de l’arbre binaire arbre. On choisit de retourner le parcours dans un tableau.



**Parcours en profondeurs : Depth First Search (DFS)**

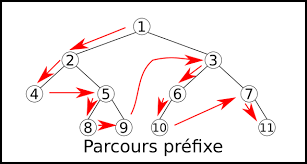
Les parcours en profondeur se définissent de manière récursive sur les arbres. Au nombre de trois :

* **Parcours en préfixe ;**
* **Parcours en postfixe ou suffixe ;**
* **Parcours en infixe ou symétrique.**

**Remarque :** Ses parcours préfixe, infixe et suffixe se distinguent par l'ordre dans lequel sont faits les traitements.

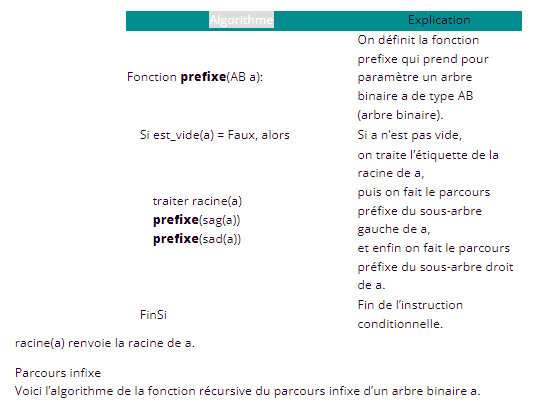
**Parcours préfixe**

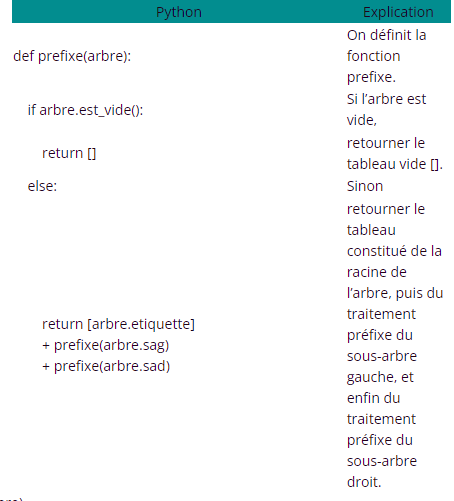
Le nœud racine est traité au premier passage avant le parcours des sous-arbres gauche et droite.



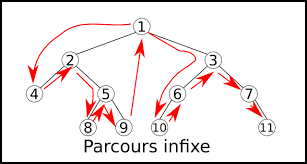
Le DFS préfixe dans l’ordre des nœuds est : **[1,2,4,5,8,9,3,6,10,7,11]**

***Astuce pro*** *: commencer à partir du nœud racine, noter chaque fois qu’on tombe sur un nouveau nœud de sous-arbre gauche vers la droite.*

**Algorithme**

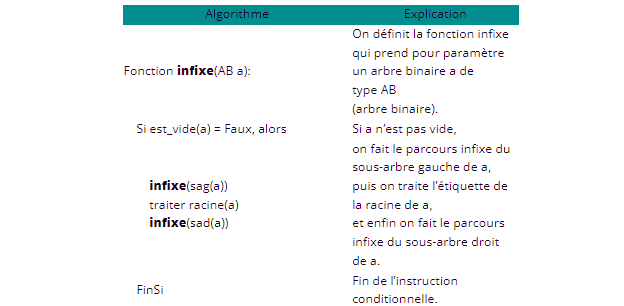
**Programmation**

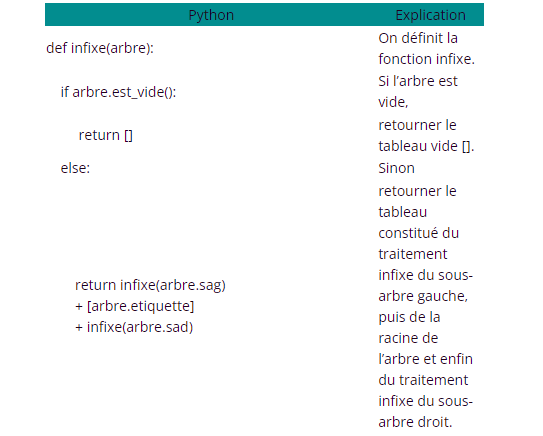
**Parcours Infixe**

Le parcours en profondeur infixe ou symétrique d'un arbre consiste à parcourir son sous-arbre gauche, puis sa racine, puis son sous-arbre droit.

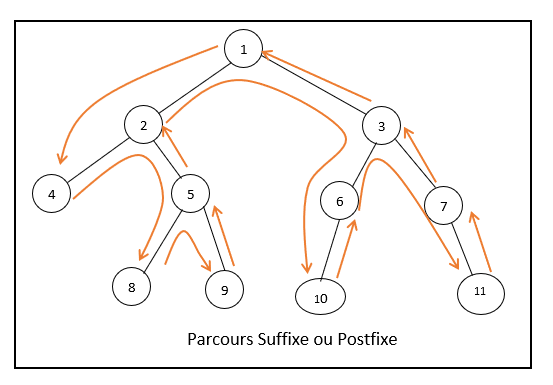
Le DFS infixe de l’arbre ci-haut est : **[4,2,8,5,9,1,10,6,3,7,11]**

Astuce pro : A partir de la racine, s’interroger, est-ce qu’on peut descendre à gauche : sinon on note la valeur du nœud en parcourant les sous-arbres gauche et droit.

**Algorithme**

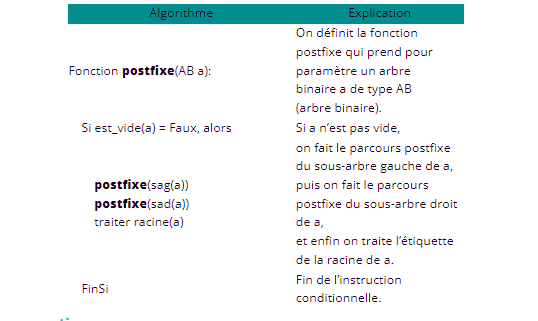
**Programmation**

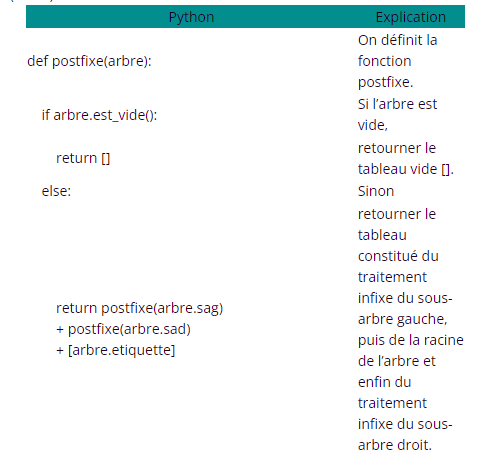
**Parcours postfixe**

Le parcours en profondeur postfixe ou suffixe d'un arbre consiste à parcourir son sous-arbre gauche, puis son sous-arbre droit, puis sa racine.

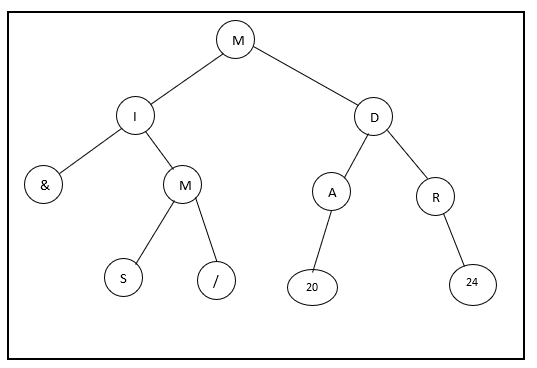
Le DFS suffixe de l’arbre ci-haut est : **[4,8,9,5,2,10,6 ,11,7,3,1]**

***Astuce pro*** *: Commencer à la racine et vérifier en notant le nœud, si à partir de celui-ci, on ne peut plus descendre ni à gauche et ni à droite en explorant tous les sous-arbres de gauche à droite*.

****Algorithme**

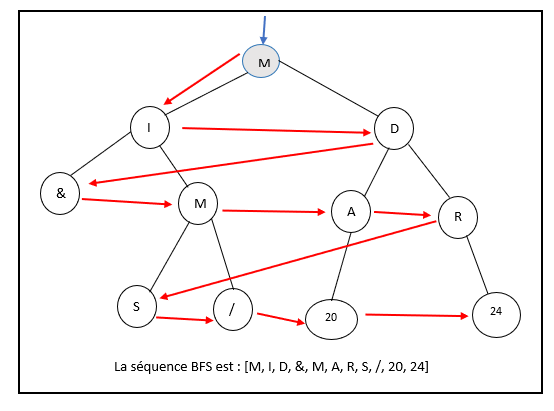
**Programmation**

**Exemples d’applications**

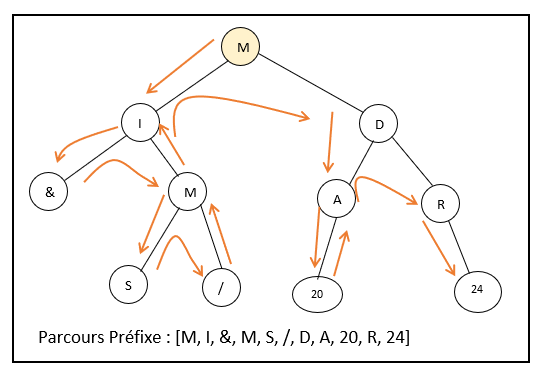
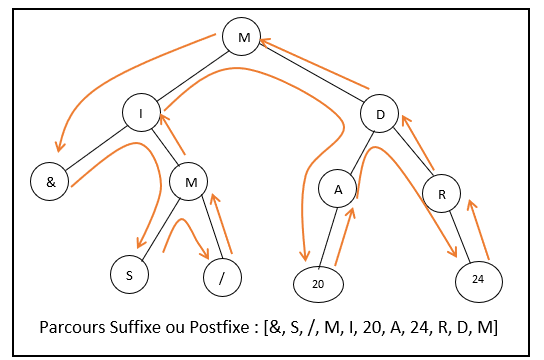
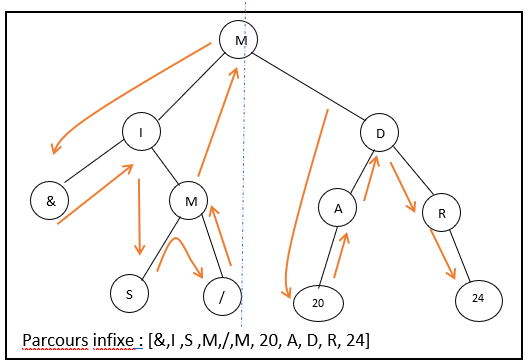
Cas d’arbre binaire :

On considère l’arbre binaire ci-dessus. Donner les parcours BFS et DFS

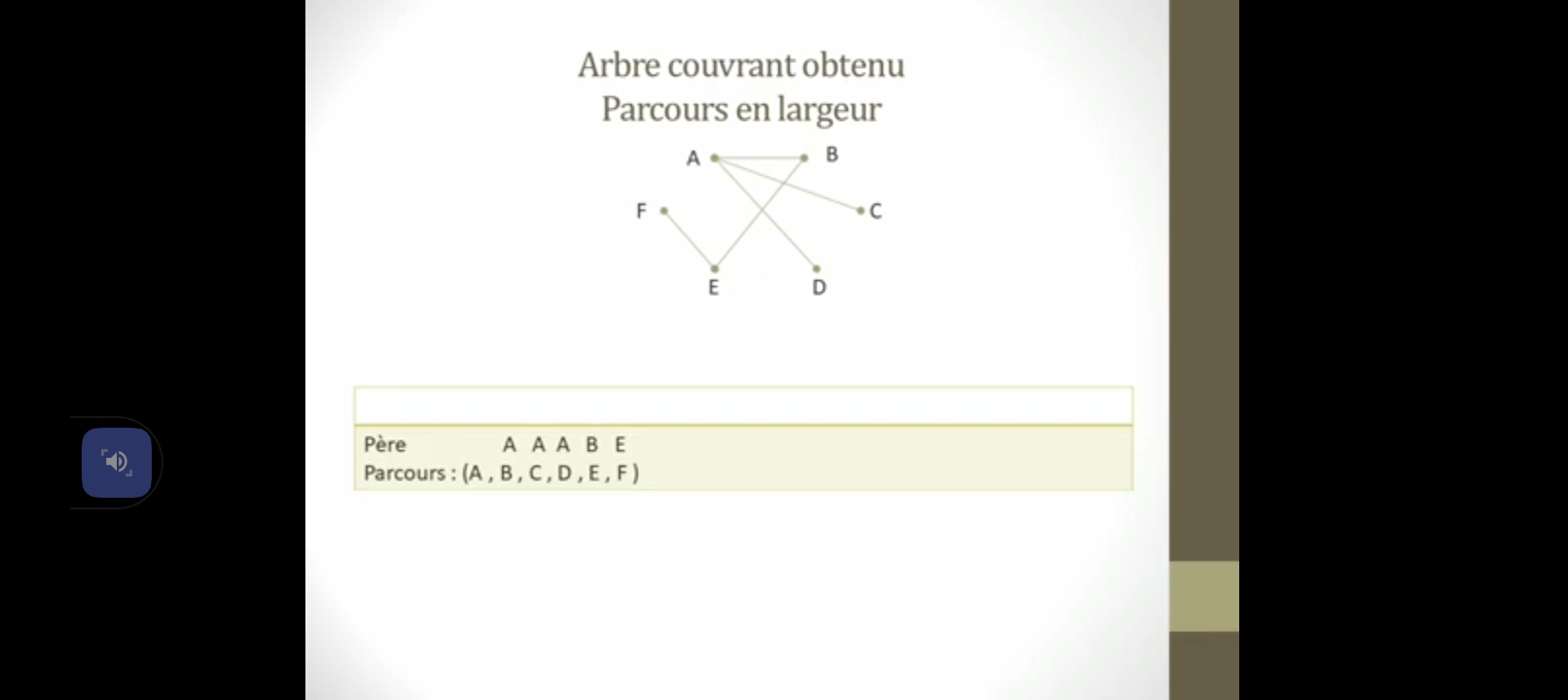
**Solution :**

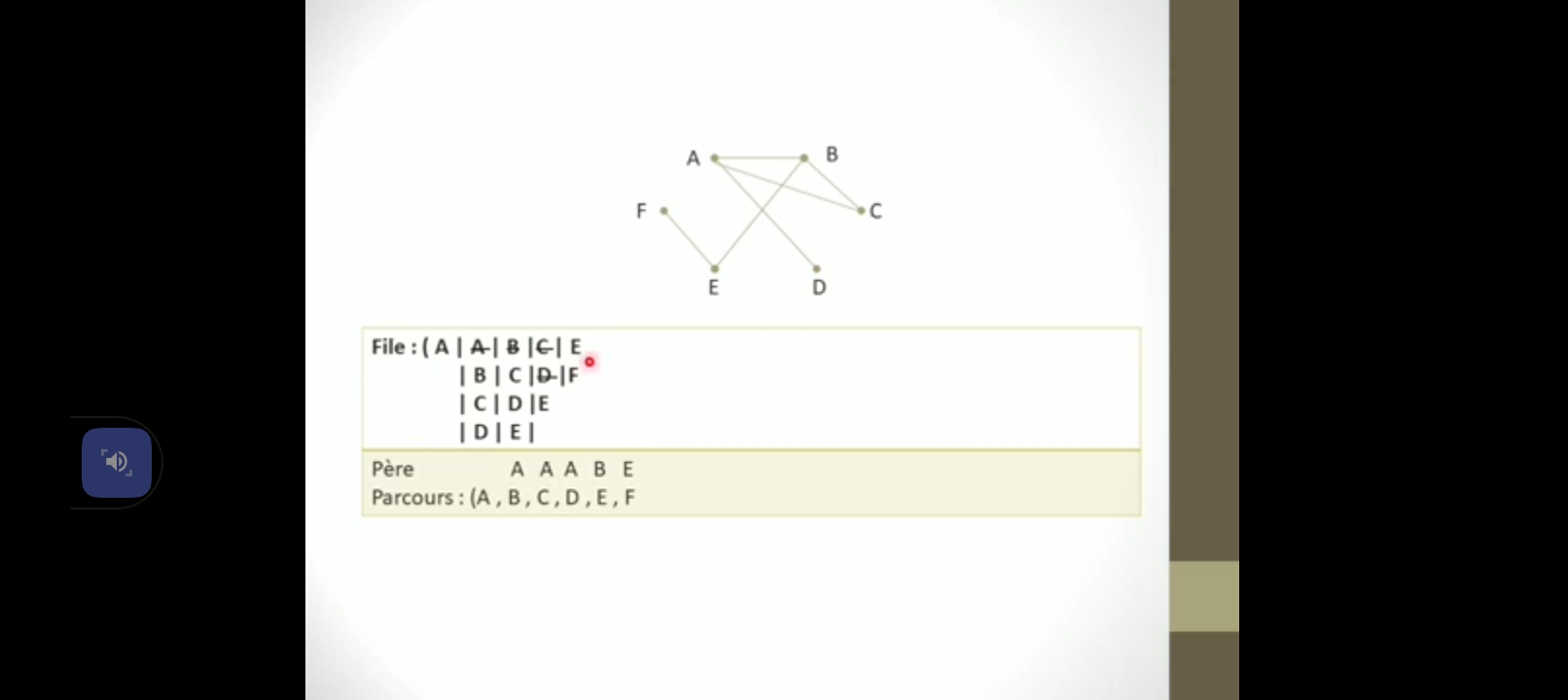
* P**arcours Itératif :**

Parcours en profondeur DFS : préfixe-postfixe- infixe

* **Parcours préfixe**
* **Parcours Suffixe**
* **Parcours Infixe**

**Cas d’un graphe**

****

**Approche de solution**

**Codage en Python**

.

class ArbreBinaire:

            def \_\_init\_\_(self, etiquette = None, sag = None, sad = None):

                self.etiquette = etiquette

                self.sag = sag

                self.sad = sad

            def est\_vide(self):

                if self.etiquette is None:

                       return True

                else:

                       return False

        # Définition de l'arbre :

        #       1

        #     /    \

        #    2      3

        #   / \    / \

        #  4   5  6   7

        #     / \

        #    8   9

                a = ArbreBinaire()

                a.etiquette = 1

                a.sag = ArbreBinaire(2)

                a.sad = ArbreBinaire(3)

                a.sag.sag = ArbreBinaire(4, ArbreBinaire(), ArbreBinaire())

                a.sag.sad = ArbreBinaire(5)

                a.sad.sag = ArbreBinaire(6, ArbreBinaire(), ArbreBinaire())

                a.sad.sad = ArbreBinaire(7, ArbreBinaire(), ArbreBinaire())

                a.sag.sad.sag = ArbreBinaire(8, ArbreBinaire(),ArbreBinaire())

                a.sag.sad.sad = ArbreBinaire(9, ArbreBinaire(),ArbreBinaire())

# Définition du fonction de parcours en largeur

                def parcours\_largeur(arbre):

                        file = []

                        file.append(arbre)

                        parcours = []

                        while len(file) > 0:

                                parcours.append(file[0].etiquette)

                                sous\_arbre = file.pop(0)

                                if not sous\_arbre.sag.est\_vide():

                                        file.append(sous\_arbre.sag)

                                if not sous\_arbre.sad.est\_vide():

                                        file.append(sous\_arbre.sad)

                        return parcours

# Parcours en Profondeurs DFS

# Définition du fonction DFS préfixe

                def prefixe(arbre):

                        if arbre.est\_vide():

                                return []

                        else:

                                return [arbre.etiquette] + prefixe(arbre.sag) + prefixe(arbre.sad)

 # Définition du fonction DFS infixe

                def infixe(arbre):

                        if arbre.est\_vide():

                                return []

                        else:

                                return infixe(arbre.sag) + [arbre.etiquette] + infixe(arbre.sad)

# Définition du fonction DFS suffixe

                def postfixe(arbre):

                        if arbre.est\_vide():

                                return []

                        else:

                                return postfixe(arbre.sag) + postfixe(arbre.sad) + [arbre.etiquette]

# Appel de fonction d'affichage

                print("parcours largeur BFS :")

# A vous d’implémenter la fonction affichage

                print("parcours en profondeur DFS :")

CONCLUSION

Bref, les parcours permettent notamment de générer des représentations linéaires des arbres et d’effectuer une recherche dans une structure arborescente. Ils sont catégorisés en deux types : le parcours en largeur dont la stratégie d’exploration est itérative avec l’utilisation d’une file et les parcours en profondeurs, qui proposent une solution récursive avec fondé sur l’ordre de traitement de la racine.

Références

<https://pixees.fr/informatiquelycee/n_site/nsi_term_structDo_liste.html>

<https://www.schoolmouv.fr/cours/arbres-et-structure-de-donnees/fiche-de-cours>

<https://www.maxicours.com/se/cours/parcourir-un-arbre-binaire/#:~:text=Le%20parcours%20en%20profondeur%20infixe,arbre%20droit%2C%20puis%20sa%20racine>.