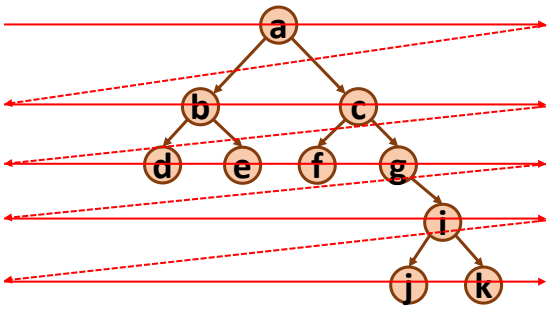
Algorithmes sur les arbres binaires

Un arbre est une structure de données hiérarchique constituée de nœuds. Avec les arbres binaires de recherche, nous avons vu une utilité de ce type de structure de données pour chercher rapidement des données dans une collection. Il y a bien d'autres utilisations des arbres, dès que les données sont organisées de manière hiérarchique. Il peut être utile de parcourir un arbre binaire, c'est-à-dire lister tous les nœuds qui le constituent. Il existe plusieurs manières de le faire, et nous allons étudier plusieurs algorithmes de parcours d'un arbre binaire.

I./ Parcours en largeur :

Dans le parcours en largeur, l'idée est de parcourir l'arbre niveau par niveau, de la gauche vers la droite :



Dans l'exemple ci-dessus, le parcours en largeur donnerait la liste suivante : [**a**, **b**, **c**, **d**, **e**, **f**, **g**, **i**, **j**, **k**].

Description de la méthode : pour ce type de parcours de l'arbre, on va utiliser une file (structure de données linéaire de type FIFO vue précédemment). On stocke l'arbre complet dans la file. On défile le premier élément (ici l'arbre complet, de racine **a**), on récupère l'étiquette de la racine (**a**) et on la stocke dans la liste. On extrait ensuite le sous-arbre gauche (si ce dernier n'est pas vide) et on le place dans la file. On extrait ensuite le sous-arbre droit (si ce dernier n'est pas vide) et on le place dans la file. On défile ensuite les arbres présents dans la file et on recommence les opérations précédentes, jusqu'à ce que la file soit vide.

Implémentation en Python :

```
def parcours_largeur(T):
    resultat = []
    if not(est_vide(T)):
        F = file_vide()
        F = push(T, F)
        while not(file_est_vide(F)):
            depile = pop(F)
            arbre = depile[0]
            F = depile[1]
            resultat.append(etiquette(arbre))
            arbre_sag = gauche(arbre)
            if not(est_vide(arbre_sag)):
                F = push(arbre_sag, F)
            arbre_sad = droit(arbre)
            if not(est_vide(arbre_sad)):
                F = push(arbre_sad, F)
    return resultat
```

Pour pouvoir utiliser cette fonction, on utilisera l'implémentation où la représentation des arbres binaires se fait grâce à des listes imbriquées (cf. cours sur les arbres binaires). Avec cette méthode, la représentation de l'arbre binaire donné plus haut en exemple serait :

```
['a',['b',['d',[],[]],['e',[],[]]],['c',['f',[],[]],['g',[],['i',['j',[],[]],['k',[],[]]]]]]
```

La fonction « **`parcours_largeur(T)`** » attend un arbre de ce type en argument et retourne la liste des sommets parcourus dans le sens indiqué dans la méthode.

Pour information, nous avons implémenté la file FIFO à l'aide d'une structure récursive selon la méthode suivante (vue précédemment dans le cours sur les files et les piles) :

```
def file_vide():
    return []

def file(valeur, file):
    return [valeur, file]

def push(valeur, file):
    if file_est_vide(file):
        return [valeur, file_vide()]
    return (file[0], push(valeur, file[1]))

def pop(file):
    return (file[0], file[1])

def file_est_vide(file):
    return file == file_vide()
```

La fonction « **file_est_vide()** » a été nommée ainsi pour ne pas être confondue avec la fonction « **est_vide()** » qui s'applique aux arbres.

Exercice : comment modifier la fonction « **parcours_largeur(T)** » pour explorer l'arbre de droite à gauche ?

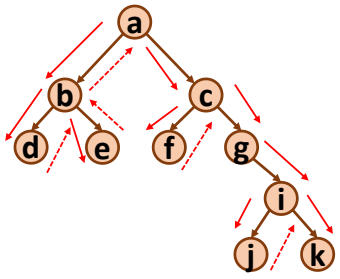
Remarque : dans l'algorithme précédent, on remarque qu'on ne met rien dans la file lorsque le sous arbre considéré est vide. Reprendre la classe « **ABR** » (vue précédemment) qui implémentait les arbres binaires de recherche, et étudier la méthode « **affiche** ». Quelles sont les similitudes et les différences avec la fonction « **parcours_largeur(T)** » ?

II./ Parcours en profondeur :

Dans le parcours en profondeur, l'idée est de parcourir l'arbre du haut vers le bas. Il existe plusieurs façons de parcourir l'arbre en profondeur :

1./ Parcours préfixe :

Dans le parcours préfixe, on place dans la liste des nœuds l'étiquette du nœud « racine » de l'arbre. Puis on traite de la même manière de façon récursive le sous-arbre gauche, puis le sous-arbre droit.



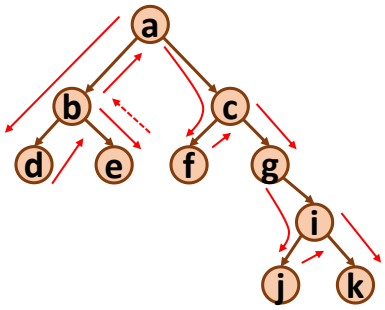
Dans l'exemple ci-dessus, on place donc l'étiquette de la racine de l'arbre dans la liste qui est **a**. On considère alors le sous-arbre gauche, et on place l'étiquette de sa racine, qui est **b** dans la liste. On considère son sous-arbre gauche, et on place l'étiquette de sa racine, qui est **d**, dans la liste. Le nœud **d** n'ayant ni sous-arbre gauche, ni sous-arbre droit, on remonte au nœud **b** et on considère alors son sous-arbre droit, et on place l'étiquette de sa racine, qui est **e**, dans la liste. Le nœud **e** n'ayant ni sous-arbre gauche, ni sous-arbre droit, on remonte au nœud **b**, dont le traitement est fini, puis au nœud **a**. On considère alors son sous-arbre droit, dont l'étiquette de la racine est **c**, que l'on place dans la liste. Le parcours continue de la même manière et donne la liste suivante : [**a**, **b**, **d**, **e**, **c**, **f**, **g**, **i**, **j**, **k**].

Implémentation en Python :

```
def parcours_profondeur_prefixe(T):
    liste = []
    if not(est_vide(T)):
        liste = liste + [etiquette(T)]
        liste = liste + parcours_profondeur_prefixe(gauche(T))
        liste = liste + parcours_profondeur_prefixe(droit(T))
    return liste
```

2./ Parcours infixe :

Dans le parcours infixe, on traite d'abord de façon récursive le sous-arbre gauche, puis le nœud racine, puis le sous-arbre droit.

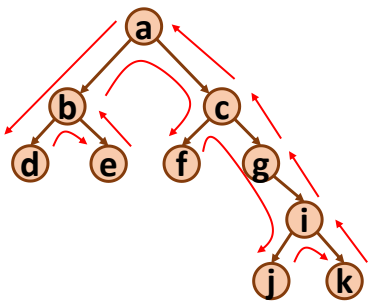


Dans l'exemple ci-dessus, on part de la racine de l'arbre et on considère le sous-arbre gauche, dont le nœud racine est **b**. On considère alors son sous-arbre gauche, dont le nœud racine est **d**. Le nœud **d** n'ayant pas de sous-arbre gauche, on place son étiquette dans la liste, puis comme il ne possède pas de sous-arbre droit, on remonte au nœud **b**, dont on place l'étiquette dans la liste, avant d'examiner son sous-arbre droit, dont le nœud racine est **e**. Le nœud **e** n'a pas de sous-arbre gauche, donc on place son étiquette dans la liste, puis comme il ne possède pas de sous-arbre droit, on remonte au nœud **b** qu'on a fini de traiter. On remonte donc au nœud **a**, dont on place l'étiquette dans la liste, avant de considérer son sous-arbre droit, dont le nœud racine est **c**. On examine alors son sous-arbre gauche, dont la racine est **f**. Comme le nœud **f** n'a pas de sous-arbre gauche, on place son étiquette dans la liste, puis on remonte au nœud **c**, dont on place l'étiquette dans la liste, avant de considérer son sous-arbre droit, et ainsi de suite, jusqu'à obtenir la liste suivante : [**d**, **b**, **e**, **a**, **f**, **c**, **g**, **j**, **i**, **k**].

Exercice : implémenter la fonction « **`parcours_profondeur_infixe(T)`** ».

3./ Parcours suffixe :

Dans le parcours suffixe, on traite d'abord de façon récursive le sous-arbre gauche, puis le sous-arbre droit, puis le nœud racine.



Dans l'exemple ci-dessus, on part de la racine de l'arbre et on considère le sous-arbre gauche, dont le nœud racine est **b**. On considère alors son sous-arbre gauche, dont le nœud racine est **d**. Le nœud **d** n'ayant pas de sous-arbre gauche, ni de sous-arbre droit, on place son étiquette dans la liste, puis on remonte au nœud **b**, et on examine son sous-arbre droit, dont le nœud racine est **e**. Le nœud **e** n'ayant pas de sous-arbre gauche, ni de sous-arbre droit, on place son étiquette dans la liste, puis on remonte au nœud **b**, dont on place l'étiquette dans la liste. On remonte alors au nœud **a** et on considère son sous-arbre droit, dont le nœud racine est **c**. On considère alors son sous-arbre gauche, dont le nœud racine est **f**. Le nœud **f** n'ayant pas de sous-arbre gauche, ni de sous-arbre droit, on place son étiquette dans la liste, puis on remonte au nœud **c**, etc… On obtient alors la liste suivante : [**d**, **e**, **b**, **f**, **j**, **k**, **i**, **g**, **c**, **a**].

Exercice : implémenter la fonction « **`parcours_profondeur_suffixe(T)`** ».

III./ POO : Amélioration de nos classes « Arbre_binaire » et « ABR » :

Nous avons précédemment créé deux classes : une classe « **`Arbre_binaire`** » et une classe « **`ABR`** ». Modifiez ces deux classes en créant les méthodes publiques permettant d'effectuer les quatre types de parcours d'arbres vues dans ce cours.

Attention : nous avons vu que l'algorithme de parcours en largeur utilisait une autre structure de données, la file. Nous avons créé une classe « File », mais si nous voulons que nos classes traitant des arbres binaires puissent être utilisées seules, il faut les rendre indépendantes de toute autre classe. Il ne faudra donc pas utiliser la classe « File ». Vous pourrez pour contourner ce problème, implémenter les fonctions relatives aux files par des méthodes privées dans vos classes sur les arbres binaires.