Arbres binaires : parcours itératifs
Largeur & Profondeur

I : Parcours récursifs préfixe, infixe et postfixe : comment transposer en itératif ?

Voici pour rappel le code d'un algorithme de parcours préfixe en récursif.

```
def parcours(arbre):
    faire_qq_chose(arbre.valeur)
    parcours(arbre.gauche)
    parcours(arbre.droit)
```

En pratique, l'appel `parcours(arbre.droit)` ne peut avoir lieu qu'une fois l'appel `parcours(arbre.gauche)` terminé.

Autrement dit, lors du parcours récursif de l'arbre binaire on ne se déplace vers la droite qu'après avoir parcouru tout le sous-arbre de gauche. Puisque ce raisonnement est valable à chaque noeud, on se déplace finalement d'abord vers le bas avant de se déplacer vers la droite. Cela correspond bien à un parcours en profondeur.

Le point important que nous avions évoqué sur la version récursive du parcours est que la pile des appels récursifs permet de gérer les "remontées" ou "retours en arrière" dans le parcours de l'arbre. Lorsque la pile d'appel diminue parce qu'un appel récursif est fini, cela revient à remonter d'un ou plusieurs nœuds dans le parcours de l'arbre pour pouvoir se diriger vers la droite (voir dernier cours si besoin).

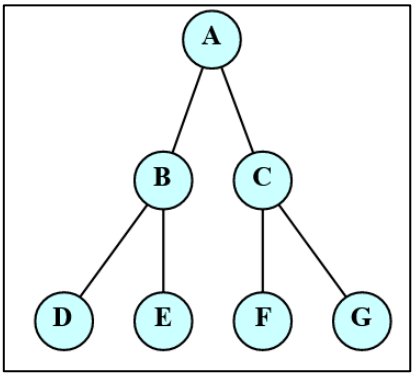
> Dans les parcours récursifs, la pile des appels récursifs joue un rôle de mémoire dans l'avancement du parcours de l'arbre. En effet c'est la pile d'appels qui mémorise à quel nœud on doit remonter lorsqu'on a fini d'explorer un sous-arbre.

Si on souhaite implémenter des algorithmes de parcours itératifs, il va bien falloir – d'une façon ou d'une autre – assurer ce rôle de mémoire explicitement (alors qu'en récursif c'est caché en arrière plan : la pile des appels est invisible du programmeur).

Cela est finalement assez simple à mettre en œuvre : plutôt que de faire deux appels récursif à `parcours(arbre.gauche)` et `parcours(arbre.droit)`, il suffit de stocker les deux sous-arbres dans une SDD en attente de pouvoir les explorer.

Lorsqu'on veut explorer un nouveau sous-arbre, il suffit alors d'aller "piocher" dans cette SDD un sous-arbre qui y a été stocké et de l'explorer à son tour après l'avoir enlevé de la SDD.

Si on fait une pioche *au hasard* pour l'arbre ci-contre on pourrait par exemple obtenir le déroulé suivant :




```
SDD = |A|
pioche de A          : SDD = ||
exploration de A     : SDD = |B|C|
pioche de B          : SDD = |C|
exploration de B     : SDD = |C|D|E|
pioche de E          : SDD = |C|D|
exploration de E     : SDD = |C|D|
pioche de C          : SDD = |D|
exploration de C     : SDD = |D|F|G|
pioche de F          : SDD = [D|G|
exploration de F     : SDD = |D|G|
pioche de D          : SDD = |G|
exploration de D     : SDD = |G|
pioche de G          : SDD = [|
exploration de G     : SDD = ||
SDD vide : fin de l'algorithme
```

On voit donc qu'on répète une succession d'étapes identiques : on extrait un nœud de la SDD puis on rajoute ses éventuels fils gauche et droit dans la SDD. *Bien entendu la pioche aléatoire présente un défaut* : on ne contrôle pas le déroulé de l'algorithme. Par ailleurs il existe des SDD parfaitement adaptées à des opérations réduites à "mettre dans SDD" et "extraire de SDD" : il s'agit par exemple des files, des piles ou des listes ☺. Ce sont donc elles que nous allons utiliser comme SDD.

Enfin, le fonctionnement des piles et files a pour conséquence intéressante que selon que la SDD est une file (FIFO) ou une pile (LIFO) on obtient un parcours en largeur ou en profondeur (ou le contraire : à vous de chercher dans le II).

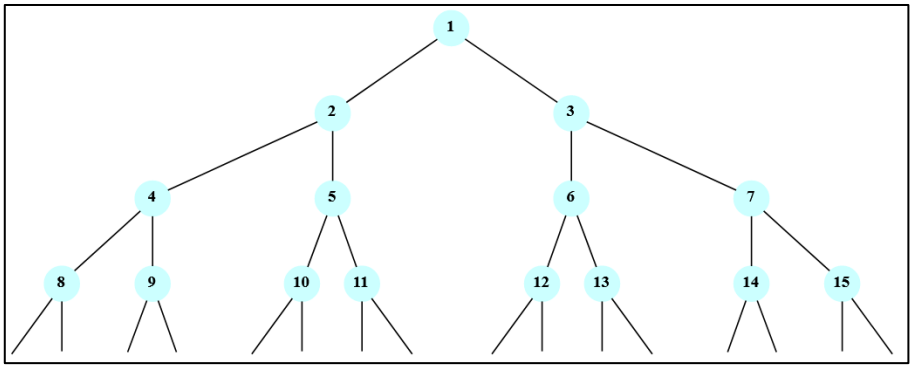
Remarque : vous allez constater que dans l'implémentation suivante on stockera les None dans la SDD (mais on peut faire autrement)

II : parcours itératifs en largeur et en profondeur

Parmi les deux algorithmes, un correspond au parcours en largeur, l'autre au parcours en profondeur. Surligner les différences entre les deux puis les appliquer scrupuleusement sur l'arbre indiqué plus bas afin de les identifier : bien garder une trace des états successifs des piles / files / listes sur la fiche donnée en Annexe (on notera ces états au niveau [*] de la boucle).

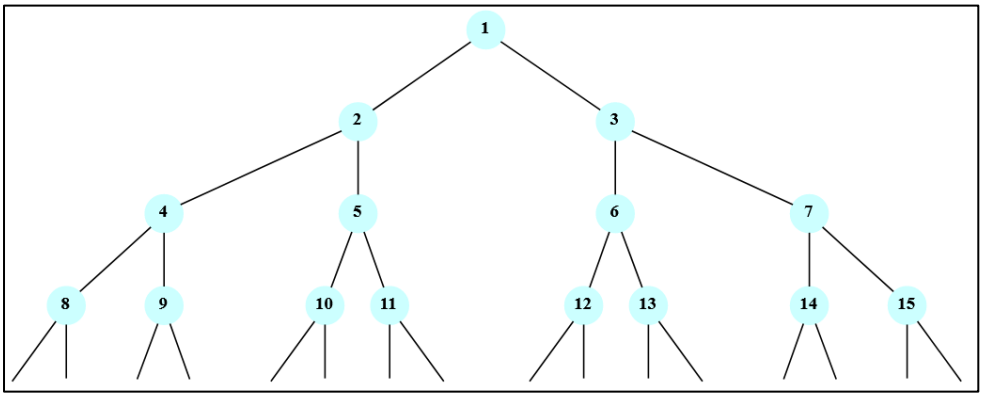
```
def parcours_1(arbre):
    sommets_visités = Liste()
    en_attente = File()

    sommets_visités.ajouter(arbre.valeur)
    en_attente.enfiler(arbre.gauche)
    en_attente.enfiler(arbre.droit)

    while not en_attente.est_vide():
        a = en_attente.défiler()
        if not a == None:
            sommets_visites.ajouter(a.valeur)
            en_attente.enfiler(a.gauche)
            en_attente.enfiler(a.droit)    [*]

    return sommets visités
```

```
def parcours_2(arbre):
    sommets_visités = Liste()
    en_attente = Pile()

    sommets_visités.ajouter(arbre.valeur)
    en_attente.empiler(arbre.droit)
    en_attente.empiler(arbre.gauche)

    while not en_attente.est_vide():
        a = en_attente.dépiler()
        if not a.est_vide():
            sommets_visités.ajouter(a.valeur)
            en_attente.empiler(a.droite)
            en_attente.empiler(a.gauche)   [*]

    return sommets visités
```

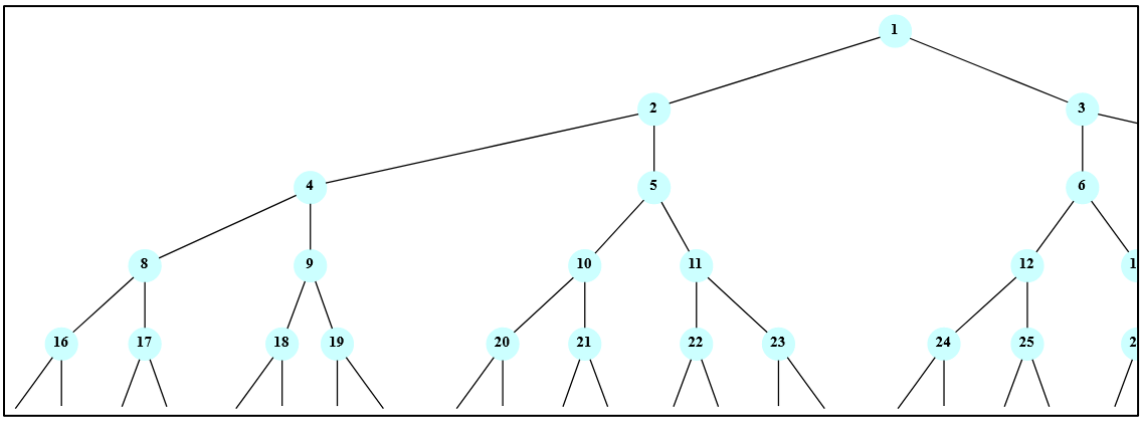


III : parcours itératifs : schémas résumant les fonctionnements des parcours

On suppose qu'on est en cours de visite d'un sommet au niveau de [*]. En couleur …… les nœuds déjà visités, en couleur …… le nœud en cours de visite, en couleur ……. les nœuds en attente dans la file/pile.



parcours 1 : Sommet en cours de visite : 6.
état de la file :



parcours 2 : Sommet en cours de visite : 18.

état de la pile :

Annexe : états successifs des piles / listes / files

les symboles ↔ et → sont là pour indiquer les entrées/sorties des listes/files/piles

parcours_1 :

sommets_visites = ↔|| #initialisation
en_attente = →||→	

sommets_visites = ↔|1| #initialisation
en_attente = →|3|2|→	

a = 2 sommets_visites = ↔|2|1| en_attente = →|5|4|3|→	
a = ... sommets_visites = ↔|...|2|1| en_attente = →|...|...|5|4|→	

parcours_2 :

sommets_visites = ↔|| #initialisation
en_attente = ↔||	

sommets_visites = ↔|1| #initialisation
en_attente = ↔|2|3|	

a = 2 sommets_visites = ↔|2|1| en_attente = ↔|4|5|3|	
a = ... sommets_visites = ↔|...|2|1| en_attente = ↔|...|...|5|3|	