Parcours d'arbres

1 Implémentation à l'aide de listes imbriquées

Contrairement à ce qui a été fait dans le TP précédent, on peut aussi impléménter une liste chaînée à l'aide de listes Python imbriquées. Un élément de liste est alors une liste qui contient une valeur et une autre liste (c'est-à-dire l'adresse de celle-ci). Par exemple, la liste chaînée `li` qui contient dans l'ordre les valeurs `1`, `2` et `3` peut se définir par simplement par l'instruction :

```
li = [1, [2, [3]]]
```

La liste chaînée vide est simplement la liste Python vide. On peut alors réécrire les fonctions de base du TP précédent : `affiche`, `taille`, `ajoute` et `suppr`. Le principe du parcours récursif est le même, mais cette fois on détecte les cas particuliers (liste vide, dernier élément) par la taille de l'élément :

```
def ajoute(li:'liste chainee', val)-> None:
    """
    Ajoute un élement de valeur val à la fin de la liste chaînée li.
    """
    if len(li) == 0: # liste vide
        li.append(val)
    elif len(li) == 1: # dernier élément
        li.append([val])
    else: # cas général : appel récursif
        ajoute(li[1], val)
```

Réécrire sur le même principe la fonction `affiche`.

On peut généraliser ce principe pour implémenter facilement des arbres : chaque noeud est une liste dont le premier élément est la valeur du noeud et les éléments suivants ses descendants. Par exemple l'arbre `T` ci-dessous :

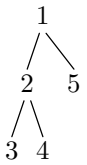

Figure 1.

sera représenté par :

```
T = [1, [2, [3], [4]], [5]]
```

Cette fois encore c'est la taille des listes qui est la clé du parcours : une liste de taille `1` représente une feuille, tandis qu'une liste de taille supérieure représente un noeud.

2 Parcours en profondeur (Depth First Search)

Contrairement aux listes chaînées, qui en tant que structures connectées à une dimension ne peuvent se parcourir que d'une seule façon (du premier au dernier élément), on peut envisager

plusieurs parcours pour les arbres. Le parcours en profondeur (DFS) consiste - en partant de la racine - à aller le plus loin possible (jusqu'à une feuille) puis ensuite à revenir en arrière :

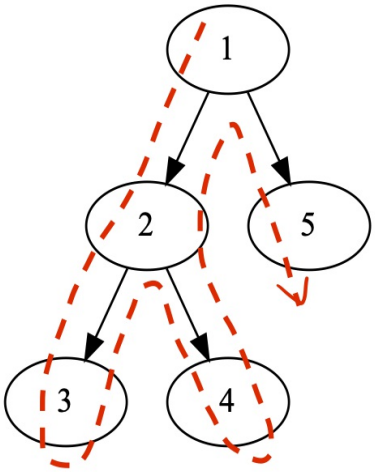

Figure 2. Un parcours DFS

Sur l'exemple précédent, l'ordre de découverte des noeuds est `1`, `2`, `3`, `4`, `5`.

Écrire une fonction récursive `DFS` qui prend une liste (un noeud) en argument et qui affiche la valeur des noeuds de son sous-arbre dans l'ordre de celui d'un parcours DFS.

3 Parcours en largeur (Breadth First Search)

Le parcours en largeur consiste à ne passer au niveau inférieur que lorsque tous les noeuds d'une hauteur donné ont été visités :

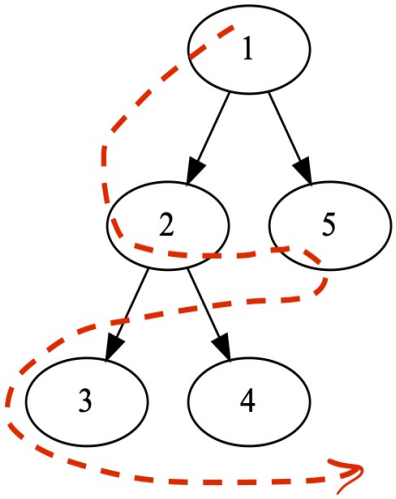

Figure 3. Un parcours BFS

Sur l'exemple d'arbre précédent, l'ordre de traitement des noeuds est donc `1`, `2`, `5`, `3`, `4`.

Écrire une fonction `BFS` qui prend une liste (un noeud) en argument et qui affiche la valeur des noeuds de son sous-arbre dans l'ordre de celui d'un parcours BFS. On utilisera une *file* pour stocker les noeuds à visiter, elle sera implémentée par une liste Python (commencer par écrire deux fonctions `enfiler` et `defiler` pour ajouter un élément à la file et le supprimer en récupérant sa valeur).

Enfin, réécrire la fonction `DFS` en utilisant le code de la fonction `BFS` et une pile à la place d'une file (commencer par écrire deux fonctions `empiler` et `depiler`).

Remarque. On peut parcourir une liste en partant de la fin sans la modifier grâce à la fonction `reversed` intégrée au langage.

Exemple. Le code ci-dessous :

```
li = [1, 2, 3]
for el in reversed(li):
    print(el)
```

Affichera `3`, puis `2`, puis `1`.