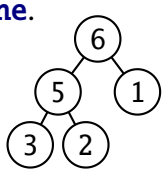
Python – Arbre binaire

Représentation des arbres

Pour modéliser les arbres, nous allons utiliser des triplets `(gauche, racine, droite)`, où `gauche` et `droite` sont également des arbres. L'arbre vide sera représenté par `None`.
Ainsi, l'arbre `a1` correspond à l'arbre ci-contre.




```
a1 = (((None, 3, None), 5, (None, 2, None)), 6, (None, 1, None))
```

Pour simplifier l'écriture, on pourra utiliser une fonction `feuille(val)` qui renvoie un arbre ne contenant qu'un seul nœud :

```
def feuille(val):
    return (None, val, None)
```

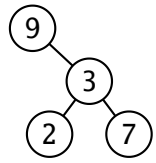
L'arbre précédent peut alors s'écrire ainsi :

```
a1 = ((feuille(3), 5, feuille(2)), 6, feuille(1))
```

Exercice 1 (sur papier) : Dessiner l'arbre correspondant à cette expression ;

```
a2 = ((None, 6, (feuille(3), 8, feuille(4))), 0, (None, 5, feuille(1)))
```

Exercice 2 (sur papier) : Donner l'expression correspondant à l'arbre ci-contre. On pourra appeler `a3` cet arbre, afin de l'utiliser dans les exercices suivant.



Fonctions de base

Afin de pouvoir manipuler et explorer les arbres, il faut utiliser les fonctions suivantes :

Fonction	Description
`racine(arbre)`	Renvoie la valeur à la racine de `arbre`.
`gauche(arbre)`	Renvoie le descendant gauche de `arbre`.
`droite(arbre)`	Renvoie le descendant gauche de `arbre`.
`est_vide(arbre)`	Renvoie un booléen indiquant si `arbre` est vide ou non.

Exercice 3 : Écrire les fonctions ci-dessus.

```
>>> racine(a1)
6
>>> racine(droite(a1))
1
>>> racine(gauche(a1))
5
>>> est_vide(gauche(a1))
False
>>> est_vide(droite(gauche(a1)))
False
>>> est_vide(gauche(droite(a1)))
True
```

Parcours en profondeur

Nous allons maintenant réaliser les différents parcours en profondeur, en affichant à chaque fois le nœud étudié. Pour rappel, voici un exemple de parcours préfixe :

```
def affichage_prefixe(arbre):
    if not est_vide(arbre):
        print(racine(arbre))
        affichage_prefixe(gauche(arbre))
        affichage_prefixe(droite(arbre))
```

```
>>> affichage_prefixe(a1)
6
5
3
2
1
```

Exercice 4 : Recopier la fonction `affichage_prefixe(arbre)` et la tester.

Exercice 5 : Écrire la fonction `affichage_infixe(arbre)` qui fait l'affichage des nœuds avec un parcours infixe.

```
>>> affichage_infixe(a1)
3
5
2
6
1
```

Exercice 6 : Écrire la fonction `affichage_suffixe(arbre)` qui fait l'affichage des nœuds avec un parcours suffixe.

```
>>> affichage_suffixe(a1)
3
2
5
1
6
```

Exploration de l'arbre

Nous allons réaliser plusieurs fonctions permettant d'analyser et d'explorer un arbre binaire.

Exercice 7 (sur papier) : Nous souhaitons réaliser une fonction récursive `taille(arbre)` qui renvoie la taille de l'arbre, c'est-à-dire le nombre de nœuds qu'il contient.

1) Quel est la taille de l'arbre vide ?
2) Si l'arbre `a` n'est pas vide, que l'arbre `gauche(a)` à une taille de `t1` et l'arbre `droite(a)` a une taille de `t2`, quelle est la taille de `a` ?

Exercice 8 : Écrire la fonction `taille(arbre)`.

```
>>> taille(a1)
5
```

Exercice 9 (sur papier) : On souhaite maintenant réaliser la fonction `hauteur(arbre)` qui renvoie la hauteur d'un arbre. Dans le cas d'un arbre vide, elle renverra `0`.

1) Si l'arbre `a` n'est pas vide, que l'arbre `gauche(a)` à une hauteur de `h1` et l'arbre `droite(a)` a une taille de `h2`, quelle est la hauteur de `a` ?
2) Si l'arbre `a` est une feuille, vérifier que la formule précédente donne le résultat attendu.

Exercice 10 : Écrire la fonction `hauteur(arbre)`. Vous pouvez utiliser la fonction `max(v1, v2)` qui renvoie le maximum des deux valeurs.

```
>>> hauteur(a1)
3
>>> hauteur(feuille(1))
1
```

Exercice 11 : Écrire une fonction `est_dans(arbre, val)` qui renvoie un booléen indiquant si `val` se trouve dans `arbre`.

```
>>> est_dans(a1, 6)
True
>>> est_dans(a1, 2)
True
>>> est_dans(a1, 4)
False
```

Exercice 12 : Écrire une fonction `maximum(arbre)` qui renvoie le maximum contenu dans l'arbre. Pour cette fonction, on suppose que toutes les valeurs sont positives ou nulles. Le maximum d'un arbre vide est 0. Vous pouvez utiliser la fonction `max(v1, v2, ..., vN)` qui renvoie le maximum d'un nombre arbitraire de valeurs.

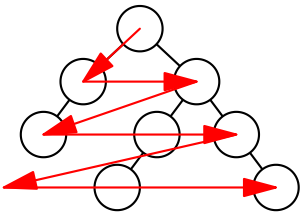
```
>>> maximum(a1)
6
>>> maximum(gauche(a1))
5
```

Exercice 13 : Compléter la fonction `maximum2(arbre)` qui renvoie le maximum contenu dans l'arbre. Cette fois-ci, on suppose que l'arbre vide n'a pas de maximum et qu'il faut lever une exception dans ce cas. Il faut donc s'assurer que les sous-arbres gauches et droites ne sont pas vides avant de faire les appels récursifs.

```
def maximum2(arbre):
    if est_vide(arbre):
        raise ValueError("Arbre vide")
    else:
        m = racine(arbre)
        if not est_vide(gauche(arbre)):
            m = max(m, ...)
        if not ...:
            m = ...
        return m
```

```
>>> maximum2(a1)
6
>>> maximum2(droite(a1))
1
```

Parcours en largeur



Les trois parcours que nous avons vu sont dit **en profondeur** car ils vont le plus en profondeur à gauche avant de traiter le moindre descendant droite. Avec un parcours **en largeur**, on va, au contraire, traiter les nœuds niveau par niveau. Ainsi, un nœud de profondeur 3 ne pourra pas être traité tant que tous les nœuds de profondeur 2 n'ont pas été traités.
Pour cela on utilise une file dans laquelle sont stockés les arbres à traiter. Au début elle ne contient que l'arbre donné en paramètre et à chaque fois qu'on retire un arbre, on rajoute les sous-arbres gauche et droit, s'ils ne sont pas vides. Lorsque la file est vide, c'est qu'on a fini le parcours de l'arbre.
Pour créer la file, on utilisera la structure `deque` de la bibliothèque `collections`. Cette structure est comme une liste, mais l'ajout et le retrait en tête ou en queue est fait en temps constant. Il faut donc rajouter la ligne suivante au début de votre fichier :

```
from collections import deque
```

L'utilisation est la suivante :

```
>>> file1 = deque()    # Création d'une file vide
>>> file1.append(6)    # Ajout de 6
>>> file1.append(3)    # Ajout de 3
>>> file1.popleft()    # On retire le premier
6
>>> file1.popleft()    # On retire le nouveau premier
3
>>> len(file1)         # Longueur de la file
0
>>> file2 = deque([7, 2, 1]) # Création à partir d'une liste
>>> file2.popleft()    # Le premier est le premier de la liste
7
```

Bien entendu, on peut mettre des arbres dans une file.
On considère la fonction ci-cdessous :

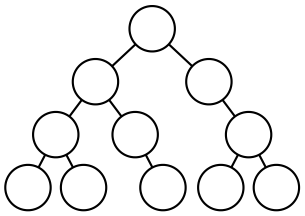
```
def parcours_en_largeur(arbre):
    if not est_vide(arbre):
        file = deque([arbre])
        while len(file) > 0:
            courant = file.popleft()
            print(racine(courant))
            if not est_vide(gauche(courant)):
                file.append(gauche(courant))
            if not est_vide(droite(courant)):
                file.append(droite(courant))
```

Exercice 14 (sur papier) :

1) Que va afficher `parcours_en_largeur(a1)` ?
2) Que va afficher `parcours_en_largeur(a2)` ?
3) Que va afficher `parcours_en_largeur(a3)` ?

Exercice 15 (sur papier) : Placer les valeurs dans les nœuds de l'arbre `a4` ci-contre pour obtenir l'affichage ci-dessous.

```
>>> parcours_en_largeur(a4)
7
1
2
3
1
7
8
9
6
4
5
```



Exercice 16 : Recopier et tester la fonction `parcours_en_largeur(arbre)`.

Exercice 17 : Écrire une fonction `est_dans_largeur(arbre, val)` qui renvoie un booléen indiquant si `val` est dans `arbre` ou pas, en faisant un parcours en largeur.

Exercice 18 : Écrire une fonction `maximum3(arbre)` qui renvoie le maximum contenu dans l'arbre. Si l'arbre est vide, il faut lever une exception.

Affichage de l'arbre

Pour pouvoir visualiser les arbres, nous allons adopter un affichage de côté. La racine sera à gauche, les feuilles à droite, les descendants gauches vers le bas et les droits vers le haut.

Exercice 19 : Écrire la fonction `affichage_cote(arbre, n=0)` qui affiche l'arbre `arbre` de profondeur `n`. À chaque fois que la profondeur augmente, l'affichage est décalé de 2 espaces vers la droite. Rien n'est affiché pour l'arbre vide.

```
>>> affichage_cote(a1)
  1
6
    2
  5
    3
```

Meilleur affichage

Pour pouvoir faire un meilleur affichage, il faut savoir quelles branches se trouvent au dessus de chaque nœud, dans quel sens et à quel endroit. Pour cela, on va associer à chaque nœud les instructions qui permettent de l'atteindre. Ces instructions seront des textes contenant des `"g"` et `"d"` indiquant s'il faut aller à droite ou à gauche. Pour la racine, le texte est vide. Par exemple, dans l'arbre affiché au dessus, pour le nœud contenant 2, il faut faire `"gd"` alors que pour trouver 1, il suffit de faire `"d"`. La fonction suivante permet de renvoyer le chemin permettant d'arriver à la valeur indiquée.

```
def trouve_val(arbre, val):
    if est_vide(arbre):
        raise IndexError("Valeur non trouvée")
    elif racine(arbre) == val:
        return ""
    else:
        try:
            return "g"+trouve_val(gauche(arbre), val)
        except:
            return "d"+trouve_val(droite(arbre), val)
```

```
>>> trouve_val(a1, 2)
'gd'
>>> trouve_val(a1, 1)
'd'
```

Il faut que la valeur soit dans l'arbre, sinon une erreur est levée. Si la valeur apparaît plusieurs fois, c'est la première occurence trouvée qui est renvoyée.

Exercice 20 : Écrire une fonction `suivre_instructions(arbre, instr)` qui renvoie la valeur du nœud trouvé en suivant les instructions données. Vous pouvez faire une fonction itérative ou récursive. Dans ce cas, vous pouvez rajouter un paramètre `i` qui vaut `0` par défaut. On suppose que les instructions sont correctes et correspondent bien à une valeur dans l'arbre.

```
>>> suivre_instructions(a1, "gd")
2
>>> suivre_instructions(a1, "d")
1
```

Afin de déterminer les branches à dessiner avant d'afficher un nœud, nous allons utiliser la fonction suivante :

```
def faire_prefixe(instr):
    res = ""
    for i in range(1, len(instr)):
        if instr[i-1] != instr[i]:
            res += "\u2502 "
        else:
            res += "  "      # Mettre 2 espaces
    if len(instr) > 0:
        if instr[-1] == "g":
            res += "\u2514\u2500"
        else:
            res += "\u250C\u2500"
    return res
```

```
>>> faire_prefixe("gd")
'│ ┌─'
>>> faire_prefixe("ggg")
'  └─'
>>> faire_prefixe("gdd")
'│  ┌─'
```

Les caractères spéciaux utilisés permettent de tracer des traits qui se joignent bien.

Exercice 21 : Écrire la fonction récursive `affichage_cote2(arbre, instr="")` qui affiche l'arbre auxquel on arrive en suivant les instructions. Si l'arbre est vide, on n'affiche rien, sinon on affiche le sous-arbre droit, la racine et le sous-arbre gauche. Pour afficher la racine, il faut d'abord afficher le prefixe obtenu avec la fonction `faire_prefixe`. Pour convertir la valeur de la racine en texte, il faut faire `str(racine(arbre))`.

```
>>> affichage_cote2(a1)
┌─1
6
│ ┌─2
└─5
  └─3
>>> affichage_cote2(feuille(4),"ggd")
  │ ┌─4
```

Pour aller plus loin

Exercice 22 : Écrire une fonction `parfait(val, h)` qui renvoie un arbre parfait de hauteur `h` et dont tous les nœuds valent `val`.

```
>>> parfait(0, 3)
(((None,0,None), 0, (None,0,None)), 0, ((None,0,None), 0, (None,0,None)))
>>> affichage_cote2(parfait(0, 4))
      ┌─0
    ┌─0
    │ └─0
  ┌─0
  │ │ ┌─0
  │ └─0
  │   └─0
0
  │   ┌─0
  │ ┌─0
  │ │ └─0
  └─0
    │ ┌─0
    └─0
      └─0
```