TP sur les arbres (1): une classe pour les arbres binaires

Stéphane Grandcolas, Valentin Emiya, François Brucker - Aix-Marseille Université

L2 MPCI - Novembre 2024

Quelques recommandations méthodologiques :

- Créez un répertoire de travail dédié à ce TP et mettez-y tous vos fichiers sources, y compris les fichiers de test, sans utiliser de sous-répértoires.
- Dans VS Code, ouvrez ce répertoire comme projet pour que les tests soient identifiés.
- Structurez chaque fichier en mettant d'abord les importations, puis les définitions (classes, fonctions), puis une partie commençant par `if __name__ == "__main__":` pour vos instructions à exécuter (par exemple, des petits codes pour tester et trouver des bugs).
- La plupart des tests sont fournis : étudiez-les et utilisez-les pour valider votre code.

Objectif. L'objectif de ce TP est de créer une classe `BinaryTree` pour modéliser les arbres binaires. Cette classe sera utilisée par le TP suivant pour une application de codage/compression. On s'appuira sur la définition par récurrence :

- il existe un arbre binaire particulier appelé arbre vide.
- si a_1 et a_2 sont deux arbres binaires et e une étiquette quelconque, (e, a_1, a_2) est un arbre binaire dont la racine porte l'étiquette e, a a_1 comme fils gauche et a_2 comme fils droit.

Les objets `BinaryTree` correspondent à la racine de l'arbre et ont trois attributs

- `label` : l'étiquette du nœud, de type quelconque
- `left_child` : le fils gauche, de type `BinaryTree` ou `None`
- `right_child` : le fils droit, de type `BinaryTree` ou `None`

L'arbre vide est représenté par `None` [1].

Remarque : dans la mise en œuvre proposé avec une unique classe `BinaryTree`, on confond les notions d'arbre et de nœud (racine). Une autre solution couramment utilisée consiste à distinguer les deux notions en créant une classe pour les nœuds et une autre classe pour les arbres, cette dernière ayant un attribut désignant le nœud racine. Il ne parait pas utile de faire cette distinction dans le cadre de ces TP.

Exercice 1 (Création de la classe)**.** Dans un fichier `tree.py`, créez une classe `BinaryTree` dont le constructeur `__init__(self, label, left_child, right_child)` prend en arguments les valeurs initiales pour les 3 attributs.

Exercice 2 (Feuille)**.** Ajoutez une méthode statique `create_leaf(label)` qui renvoie une feuille portant l'étiquette `label`, et une méthode (non-statique) `is_leaf(self)` qui renvoie `True` ou `False` en fonction de si l'objet courant est une feuille ou non.

Exercice 3 (Exemples d'arbres)**.** Dans la partie `main` en bas du fichier, créez les arbres de la Figure 1 (vous ne pourrez pas les afficher graphiquement avant d'avoir fait la question suivante). Vous utiliserez ces exemples pour illustrer les fonctionnalités des exercices suivants.

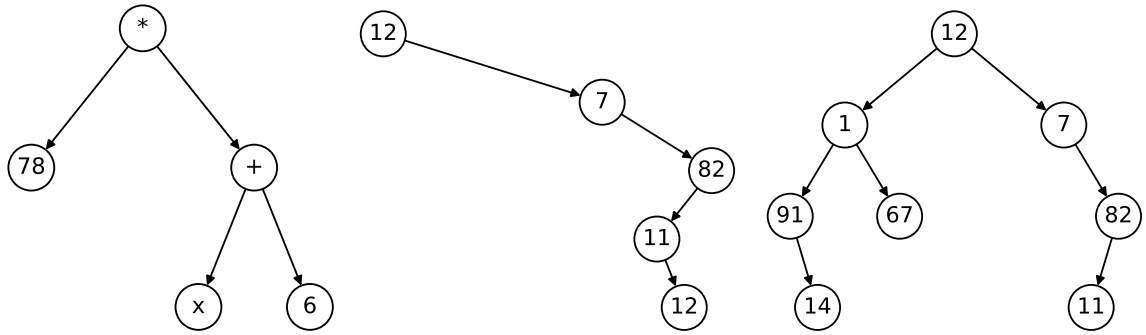

FIGURE 1 – Exemples d'arbres

Exercice 4 (Hauteur, nombre de feuilles et de nœuds)**.** Ajoutez une méthode `get_height(self)` qui renvoie la hauteur de l'arbre, une méthode `get_n_leaves(self)` qui renvoie le nombre de feuilles et une méthode `get_n_nodes(self)` qui renvoie le nombre de nœuds. Remarque : ces méthodes ne fonctionne que sur des instances de BinaryTree, donc sur des arbres non-vides.

Vous pouvez afficher les exemples en utilisant la fonction `plot_tree` de `tree_utils.py`.

Les exercices suivants sont consacrés aux différents parcours, en utilisant une fonction f appliquée à chaque nœud le long du parcours. Vous pourrez créer et utiliser une fonction qui affiche l'étiquette du nœud (voir aussi ce que fait la fonction utilisée dans les tests fournis).

Exercice 5 (Parcours en profondeur préfixe/infixe/postfixe)**.** Ajoutez des méthodes `preorder_walk(self, f)`, `inorder_walk(self, f)` et `postorder_walk(self, f)` qui effectuent un parcours en profondeur de l'arbre en appliquant la fonction `f` à chaque nœud visité de manière préfixe, infixe et postfixe, respectivement.

Exercice 6 (Parcours en largeur)**.** Le parcours en largeur examine le nœud racine, puis tous ses fils de gauche à droite, puis les petit-fils, etc. Les nœuds sont donc parcourus par profondeur croissante, de gauche à droite. Pour cela, on stocke les sommets à visiter dans une file, c'est-à-dire une structure « first-in, first-out », où l'élément que l'on extrait est l'élément inséré le plus ancien. En Python, il suffit d'utiliser une liste en insérant les éléments à la fin avec la méthode `.append` et en extrayant l'élément de tête avec la méthode `.pop(0)`. L'algorithme de parcours en largeur s'écrit :

- créer une file et y insérer la racine de l'arbre
- tant que la file n'est pas vide
  - extraire le nœud en tête de file
  - traiter le nœud
  - s'ils existent, insérer son fils gauche puis son fils droit dans la file

Ajoutez une méthode `levelorder_walk(self, f)` qui effectue un parcours en largeur de l'arbre en appliquant la fonction `f` à chaque nœud visité.

Exercice 7 (Arbre parfait)**.** Ajoutez une méthode statique `create_perfect(height)` qui renvoie un arbre parfait de hauteur `height` dont chaque nœud a `None` comme étiquette. Créez une fontion de test appropriée.

Exercice 8 (Égalité)**.** Ajoutez une méthode spéciale `__eq__(self, other)` qui teste si l'arbre courant est égal à un autre arbre `other`.

1. Attention, une autre possibilité consiste à définir l'arbre vide comme une instance de `BinaryTree` dont chaque attribut vaut `None`.