Arbres binaires - 2e partie

Terminale NSI

qkzk

décembre 2019

Un module de manipulation d'arbres binaires

La classe `BinaryTree`, définie dans le module `binary_tree.py` proposé ici, permet de représenter des arbres binaires.

Cette classe fournit

- « deux » constructeurs : `BinaryTree()` et `BinaryTree(data, left, right)`
- trois accesseurs : `get_data()`, `get_left_subtree()`, et `get_right_subtree()`
- un reconnaisseur : `is_empty()`

Essayez

```
import binary_tree as bt

help(bt.BinaryTree)
```

pour afficher l'aide associée à cette classe.

Ce module propose aussi des fonctionnalités de visualisation d'arbres binaires, et d'exportation aux formats DOT et PNG.

Parcourir les arbres

Parcourir un arbre consiste à visiter chacun des nœuds de l'arbre, à effectuer un traitement sur chacune des étiquettes des nœuds de l'arbre.

On distingue différents parcours selon l'ordre dans lequel les nœuds seront visités.

On distingue principalement les parcours en profondeur d'abord et en largeur.

Trois parcours en profondeur d'abord

Le principe des parcours en profondeur est de visiter récursivement les nœuds de l'arbre.

On distingue trois parcours, selon que l'on traite l'étiquette d'un nœud :

- avant la visite de ses fils gauche et droit

- après la visite de son fils gauche (et avant la visite de son fils droit)
- après la visite des ses fils gauche et droit

On utilise respectivement les termes

- préfixe
- infixe
- postfixe

pour désigner ces trois parcours en profondeur.

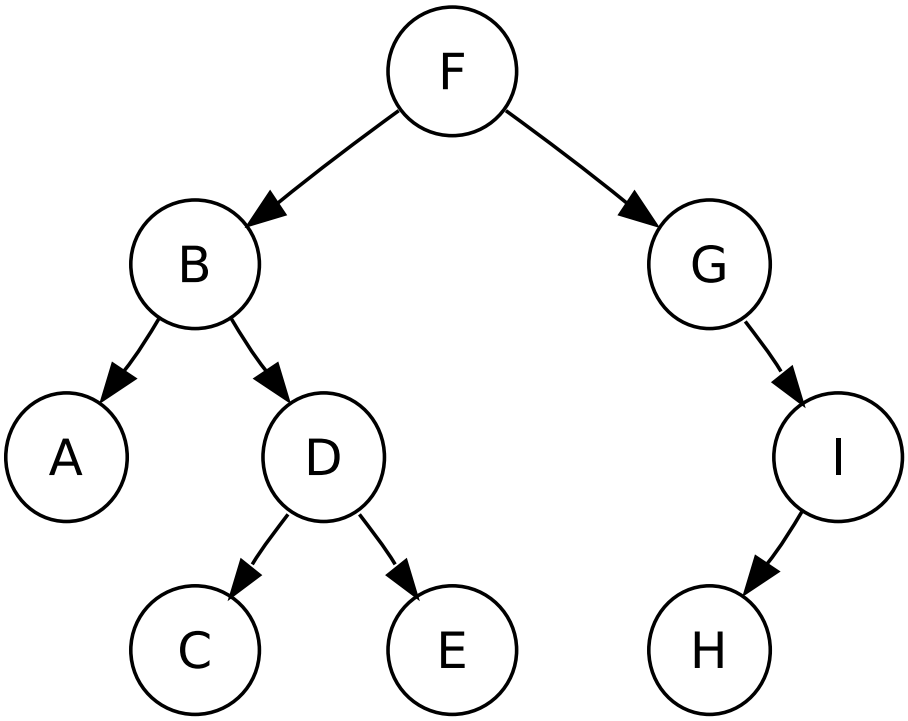
≫ Préfixe, infixe, et postfixe

Application

Donnez le résultat de l'appel de chacune des trois fonctions suivantes

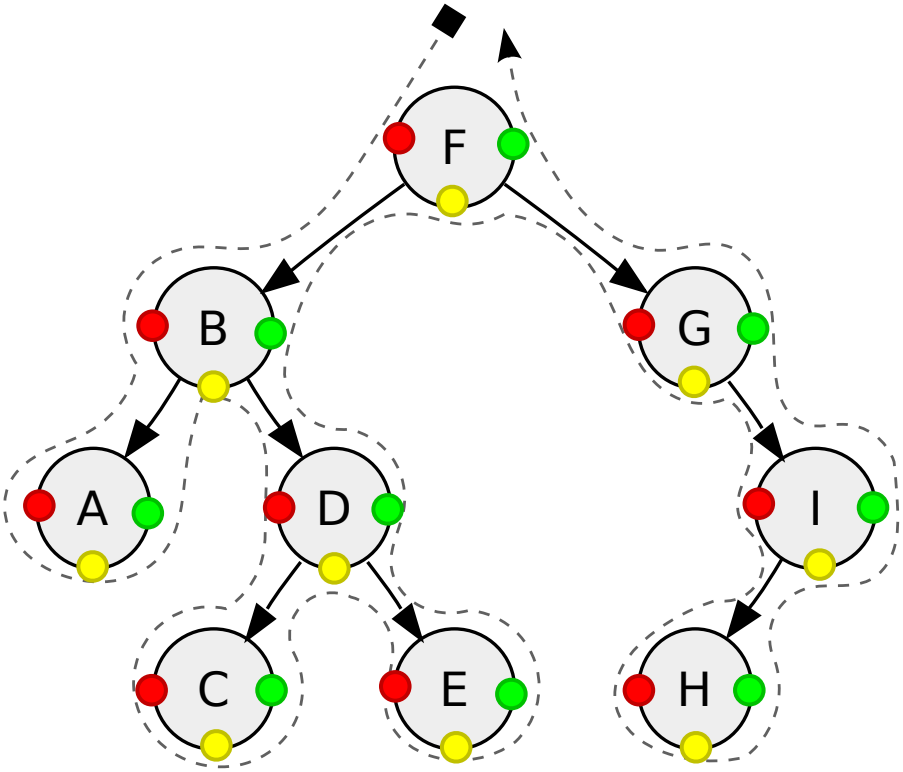
```
def afficher_A(a):
    if not a.is_empty():
        afficher_A(a.get_left_subtree())
        print(a.get_data())
        afficher_A(a.get_right_subtree())

def afficher_B(a):
    if not a.is_empty():
        afficher_B(a.get_left_subtree())
        afficher_B(a.get_right_subtree())
        print(a.get_data())

def afficher_C(a):
    if not a.is_empty():
        print(a.get_data())
        afficher_C(a.get_left_subtree())
        afficher_C(a.get_right_subtree())
```

sur l'arbre ci-dessous.



Associez le terme adéquat parmi *préfixe*, *infixe*, et *postfixe* à chacune des trois fonctions d'affichage.

Associez une des couleurs rouge, verte, jaune, à ces termes et fonctions selon le schéma suivant :

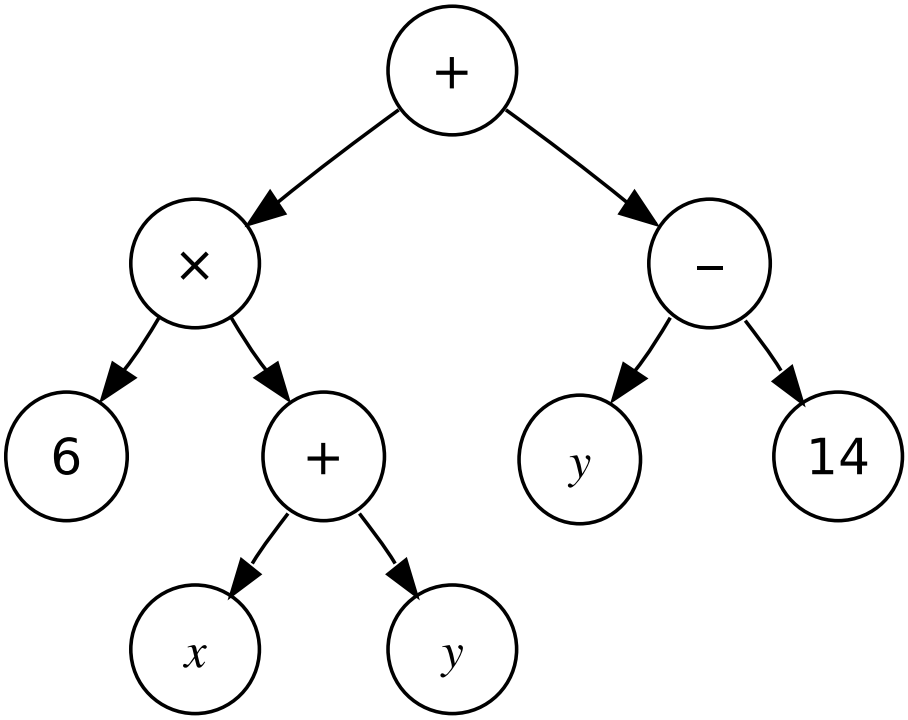


Représentation d'expressions arithmétiques

Programmation

Une expression arithmétique construite avec des opérateurs binaires – c'est-à-dire à deux opérandes telles l'addition, la soustraction, la multiplication, la division –, peut être représentée par un arbre binaire dont les nœuds internes portent les opérateurs et les feuilles des symboles de variables – x, y, z… –, ou des constantes – 6, 14, etc. –.

Ainsi, l'arbre suivant représente l'expression `$6(x+y)+(y-14)$`



Des expressions

Cet arbre, et d'autres exemples peuvent être construits avec les instructions suivantes :

```
# 6 * (x+y) + (y-14)
expression1 = bt.BinaryTree('+',
                            bt.BinaryTree('*', feuille(6), bt.BinaryTree('+', feuille('x'), feuille('y'))),
                            bt.BinaryTree('-', feuille('y'), feuille(14)))

# (x+5) * (y/2)
expression2 = bt.BinaryTree('*',
                            bt.BinaryTree('+', feuille('x'), feuille(5)),
                            bt.BinaryTree('/', feuille('y'), feuille(2)))
# 4 * (x-1)
expression3 = bt.BinaryTree('*',
                            feuille(4),
                            bt.BinaryTree('-', feuille('x'), feuille(1)))
```

≫ Imprimer une expression

Proposez une fonction qui accepte en paramètre un arbre représentant une expression arithmétique, et imprime cette expression.

≫ Évaluer une expression

Proposez une fonction qui accepte en paramètre un arbre représentant une expression arithmétique, et renvoie la valeur de cette expression.

Il est également nécessaire de fournir à cette fonction les valeurs associées à chacune des variables présentes dans l'expression. Quelle structure de données proposez-vous d'utiliser pour mémoriser ces associations ?

≫ Écriture polonaise inverse

Quel affichage produit un parcours postfixe de l'arbre représentant notre expression arithmétique ?

Remarquez que cette expression non-ambiguë ne nécessite pas de parenthèses.

Sachez que cette forme est exploitée pour produire le code machine d'évaluation de l'expression pour une machine à pile.

Parcours en largeur

Un parcours en largeur visite les nœuds d'un arbre niveau par niveau : le nœud racine de profondeur nulle, les nœuds de profondeur 1, puis les nœuds de profondeur 2, etc.
Pour une profondeur donnée, on visite les nœuds de gauche à droite.

Le principe est que lors de la visite du niveau de profondeur $`i`$, on mémorise dans une structure ad hoc les nœuds fils, qui sont donc des nœuds de profondeur $`i+1`$ qui seront à visiter une fois la visite de la profondeur $`i`$ terminée.

La structure dans laquelle mémoriser les nœuds fils doit permettre :

- d'ajouter un élément (un nœud) à la structure ;
- de récupérer un élément ; plus précisément de récupérer l'élément le plus anciennement inséré ;
- de tester que la structure est vide — notre algorithme est terminé.

On reconnaît là l'interface du type de données abstrait *file* qui met en œuvre le principe FIFO : premier arrivé, premier sorti.

Pour la réalisation de notre parcours en largeur, les éléments de cette file sont des arbres : l'arbre à parcourir, puis l'arbre fils gauche de sa racine, puis l'arbre fils droit de sa racine, etc.

≫ Parcourir en largeur

Proposez une fonction Python qui renvoie la liste des étiquettes d'un arbre binaire donné ; cette liste sera ordonnée selon un parcours en largeur d'abord de l'arbre.

≫ Parcourir avec une pile

Aller plus loin

Quel parcours de l'arbre obtient-on si on remplace la *file* utilisé dans l'algorithme précédent par une *pile*, donc une structure LIFO, dernier arrivé, dernier sorti ?

Que faut-il modifier pour obtenir un des trois classiques parcours en profondeur ?

Arbres de recherche

Notion

Un *arbre de recherche*, ABR, est un arbre binaire qui va être utilisé pour réaliser « efficacement » des opérations de recherche d'une valeur, mais aussi des opérations d'insertion et suppression de valeurs.

Les valeurs des étiquettes de l'arbre doivent donc appartenir à un ensemble ordonné.

Caractériser les arbres binaires de recherche

→ *Définition.* Arbre binaire de recherche – ABR

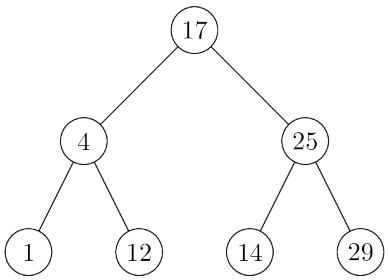
Un *arbre binaire* est un arbre binaire tel que pour tout nœud d'étiquette e :

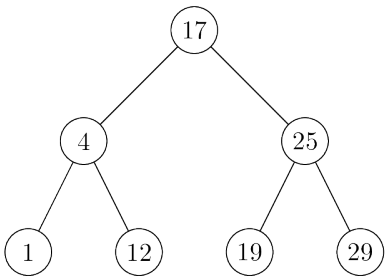
- les étiquettes de tous les nœuds de son sous-arbre gauche sont inférieures ou égales à e, et
- les étiquettes de tous les nœuds de son sous-arbre droit sont supérieures à e.

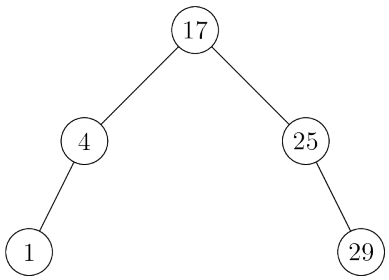
≫ Des arbres binaires de recherche

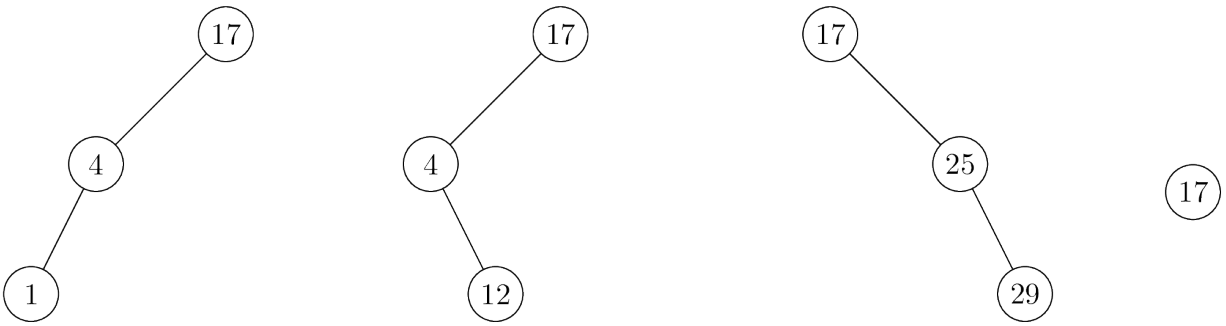
Application

Indiquez quels sont parmi les arbres suivants ceux qui sont des arbres binaires de recherche.









≫ **Les plus petite et plus grande étiquettes**

Dans quel nœud d'un arbre binaire de recherche se trouve la plus petite étiquette ?
La plus grande ?

Dans quel nœud d'un arbre binaire de recherche se trouve l'étiquette médiane ?

≫ **Parcours ordonné**

Lequel des classiques parcours d'arbres binaires permet de visiter les nœuds d'un ABR dans l'ordre des étiquettes ?

≫ **Reconnaître un arbre binaire de recherche**

Programmation

Proposez un prédicat Python qui reconnaît si un arbre binaire donné est un arbre binaire de recherche.

Deux pistes possibles pour cela sont

1. De réaliser un parcours de l'arbre qui devrait visiter les nœuds dans l'ordre croissant (exercice précédent), et vérifier que les étiquettes sont bien dans l'ordre croissant.
   (On suppose qu'il n'y a pas de doublons dans l'arbre.)
2. De proposer une fonction récursive qui renvoie la plus petite et la plus grande des valeurs des étiquettes d'un arbre, et vérifie qu'il est un ABR.

Recherche d'une valeur dans un arbre binaire de recherche

Programmation

Proposez une fonction Python qui renvoie le nœud d'un arbre binaire de recherche dont l'étiquette est égale à une valeur donnée.
La fonction pourra retourner `None` si la valeur n'était pas présente dans l'arbre.

Insertion d'une valeur dans un arbre binaire de recherche

L'insertion d'une valeur `v` dans un arbre binaire de recherche peut reposer sur le déroulé récursif suivant :

- si l'arbre est vide, renvoyer l'arbre arbre constitué d'une unique feuille portant la valeur `v`
- si la valeur à insérer `v` est inférieure ou égale à la valeur de la racine, renvoyer l'arbre formé de
  - la racine
  - l'arbre formé du fils gauche dans lequel aura été inséré la valeur `v`
  - le fils droit

- sinon – la valeur `v` est supérieur à la valeur de la racine –, renvoyer l'arbre formé de
  - la racine
  - le fils gauche
  - l'arbre formé du fils droit dans lequel aura été inséré la valeur `v`

≫ **Insérer une valeur dans un ABR**

Proposez une fonction Python qui renvoie un arbre binaire de recherche donné augmenté d'une valeur donnée.

Suppression d'une valeur dans un arbre binaire de recherche

Soit une valeur `v`, nous cherchons à supprimer, s'il exsite, le nœud d'étiquette `v` dans un arbre binaire de recherche donné.

Passons en revue les différents cas de figure :

- si l'arbre est vide, il n'y a rien à faire
- si l'arbre est réduit à une feuille, selon qu'elle porte cette valeur `v` ou non, il s'agit de renvoyer l'arbre vide ou la feuille
- si la valeur `v` est inférieure à la racine, à la manière de ce que nous avons fait pour l'insertion, reconstruire l'arbre formé de
  - la racine,
  - le fils gauche amputé de la valeur `v`
  - le fils droit
- si la valeur est supérieure à la racine, on agit de même, symétriquement pour les fils gauche et droit
- si la racine porte la valeur `v` et ne possède qu'un unique fils, ce nœud fils devient le nouvel arbre

Reste à traiter la cas où la racine porte la valeur `v` et possède deux fils.

- considérons le *successeur* de la racine ; il s'agit du nœud portant la plus petite des valeurs supérieures à `v`
- par définition d'un ABR, ce successeur est le nœud le plus à gauche du fils droit de la racine
- l'arbre amputé de `v` est donc formé :
  - de la valeur de ce successeur
  - du fils gauche
  - du fils droit amputé de la valeur du successeur.

Une implémentation « efficace » évite de parcourir le fils droit à la recherche de la valeur du successeur, puis de parcourir le fils droit pour supprimer le nœud portant cette valeur.

Coût des opérations

Les arbres binaires de recherche sont introduits pour répondre au besoin de réaliser avec la même efficacité les trois opérations de *recherche*, *ajout*, et *suppression* d'une valeur.

Chacun des algorithmes que nous avons définis pour ces trois opérations nécessite de visiter un arbre de sa racine à une feuille, comparant une valeur à l'étiquette des nœuds visités.
Le nombre de comparaisons sera donc égal à la *hauteur* de l'arbre.

Les parcours de la racine à une feuille peuvent être interrompus dans le cas ou le nœud recherché est un nœud interne.

L'analyse du coût de ces algorithmes se ramène dont à l'étude de la hauteur des arbres binaires des recherche.

Dans le meilleur des cas, un arbre binaire de recherche est *équilibré*. Sa hauteur est alors $\lfloor \log_2 n \rfloor$, n étant la taille, nombre d'éléments, de l'arbre.
Le coût des algorithmes pour nos trois opérations est alors au pire logarithmique.

Dans le pire des cas, un arbre binaire de recherche est *filiforme*. Sa hauteur est alors égale à $n - 1$.
Le coût des algorithmes pour nos trois opérations est alors au pire linéaire.

L'analyse du coût en moyenne des algorithmes nécessite de considérer

- la profondeur moyenne d'un nœud dans un arbre binaire de recherche quelconque,

et

- la hauteur d'un arbre « moyen » parmi tous les arbres binaires.

Nous ne mènerons pas cette étude ici.

→ *Résultat.* Complexité des algorithmes sur les arbres binaires de recherche

À retenir

La complexité des algorithmes de *recherche*, *insertion*, et *suppression* d'une valeur dans un arbre binaire de recherche est

- en moyenne logarithmique,
- au pire linéaire.

Le coût de ces opérations est au pire logarithmique dans le cas d'arbres de recherche *équilibrés*.

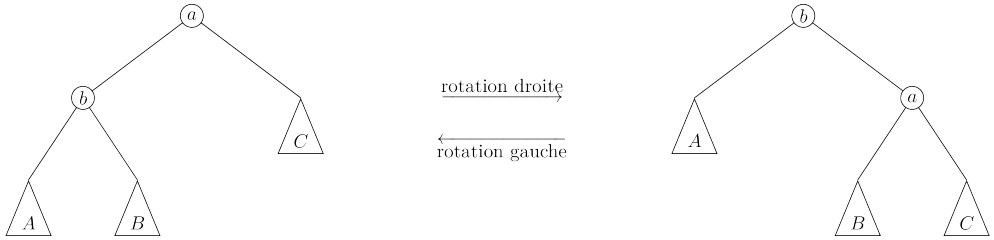
Maintenir l'équilibre

Aller plus loin

Le coût au pire logarithmique des opérations sur les arbres de recherche équilibrés motive la modification de nos algorithmes pour tenter de conserver cette propriété d'équilibre des arbres.

Le principe général repose sur des rotations opérées lors de l'insertion ou la suppression d'une valeur.

On opère des rotations simples comme celles ilustrées ci-dessous, et des rotations doubles.



Ces arbres binaires de recherche équilibrés sont nommés AVL d'après le nom de leurs inventeurs, Georgii Adelson-Velsky et Evguenii Landis en 1962.