Arbre binaire de recherche

Une structure de données doit pouvoir évoluer et bénéficier des opérations **Recherche-Ajout-Suppression** avec la meilleure efficacité possible

Or la complexité de la recherche d'un élément dans une structure linéaire est linéaire au pire, et peut être logarithmique à condition que la liste soit triée ce qui pose le problème de maintenir la liste triée après l'ajout ou la suppression d'un élément

La structure d'arbre binaire de recherche est une réponse à ces contraintes

1 Arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire avec les contraintes suivantes :

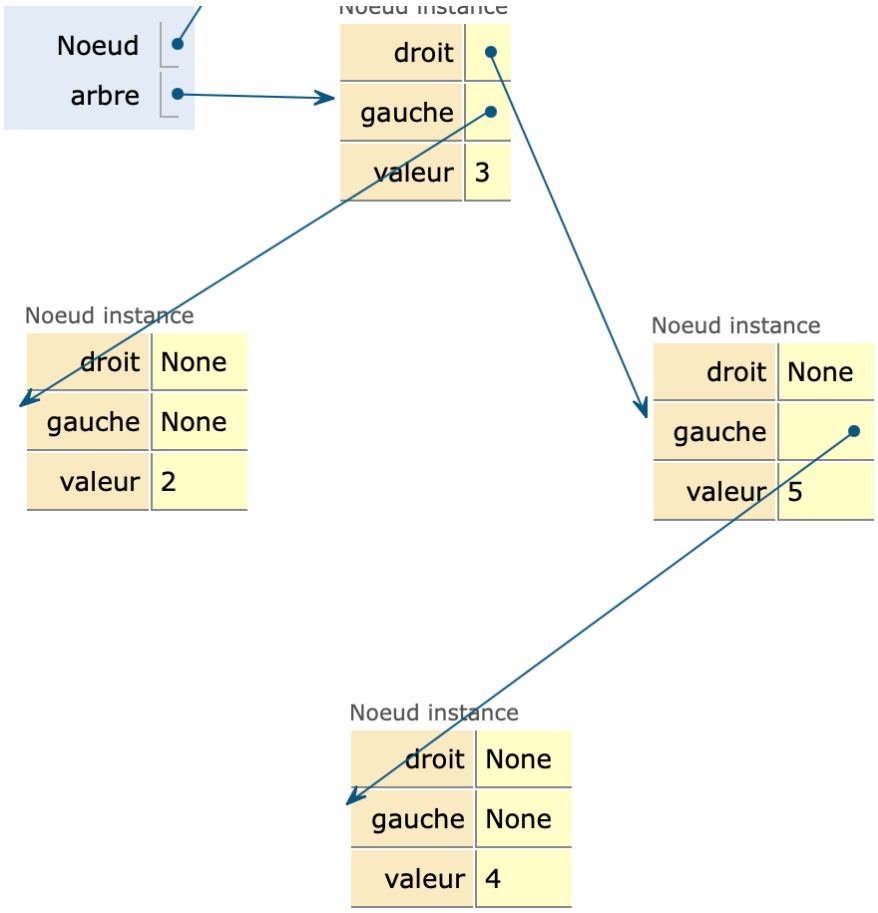
Toutes les valeurs des noeuds sont du même type et comparables, de plus

Pout tout noeud n de l'arbre

1. Tous les noeuds du sous arbre gauche de n ont une valeur **inférieure ou égale** à la valeur de n
2. Tous les noeuds du sous arbre droit de n ont une valeur **supérieure ou égale** à la valeur de n

Exemple :

On a visualisé avec Python Tutor un arbre binaire de recherche



obtenu à partir du code suivant

```
class Noeud:
    def __init__(self,v,g,d):
        self.valeur = v
        self.gauche = g
        self.droit = d

arbre = Noeud(3,Noeud(2,None,None),
Noeud(5,Noeud(4 ,None,None),None))
```

Ce n'est pas pratique de créer un arbre binaire de recherche "à la main" de cette manière, de plus on risque de se tromper

On va donc implémenter une fonction `ajouter(arbre,valeur)` qui va ajouter à un arbre binaire de recherche existant `arbre`,

la valeur `valeur` **à condition qu'elle ne s'y trouve pas**

A l'origine on part de l'arbre vide `None`

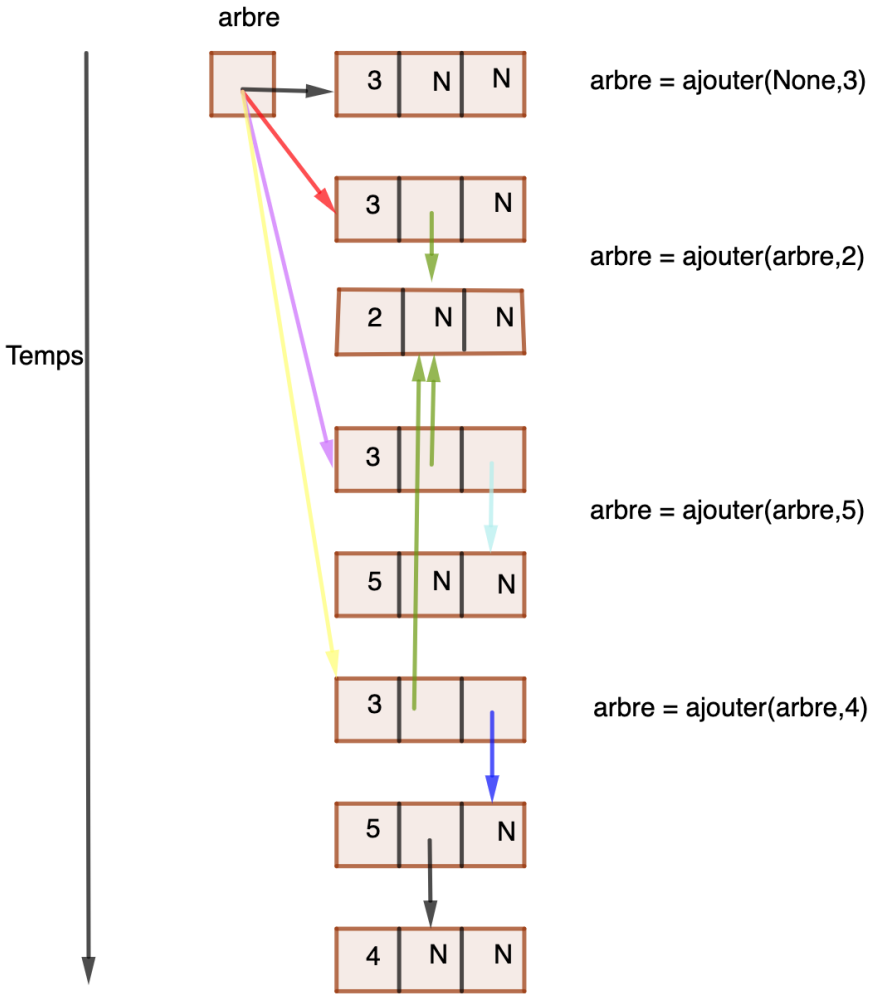
```
def ajouter(arbre,valeur):
    if arbre is None:
        return Noeud(valeur,None,None)
    if valeur < arbre.valeur:
        return Noeud(arbre.valeur,
        ajouter(arbre.gauche,valeur),arbre.droit)
    elif valeur > arbre.valeur:
        return Noeud(arbre.valeur,arbre.gauche,
        ajouter(arbre.droit,valeur))
    else:
        return arbre
```

On construit l'arbre binaire de recherche avec les appels suivants

```
arbre = ajouter(None,3)
arbre = ajouter(arbre,5)
arbre = ajouter(arbre,2)
arbre = ajouter(arbre,4)
```

Remarques

1. La fonction récursive conserve la structure d'arbre binaire de recherche mais met à jour les références qui relient les éléments entre eux au cours du temps



On voit sur le dessin que les références représentées par des flèches de différentes couleurs changent au cours du temps mais la structure est conservée

On observe qu'à chaque ajout on crée un nouveau noeud pour contenir la valeur de la racine. Le "Garbage collector" se charge d'éliminer de la mémoire les anciens noeuds

2. Si on commute les appels on n'obtiendra pas le même arbre binaire de recherche
3. Il n'y a pas de doublons dans l'arbre binaire de recherche

Maintenant on s'intéresse à la recherche d'une valeur dans un arbre binaire de recherche

Algorithme 1 : Rechercher dans un ABR

```
rechercher (A,valeur)
début
    Données : Un abr A
    Résultat : Vrai si la valeur est dans A, Faux sinon
1   si A est vide alors
2       retourner Faux
3   fin
4   si valeur < A.valeur alors
5       retourner rechercher (A.gauche,valeur)
6   fin
fin
```

Traduit en Python

```
def rechercher(arbre,valeur):
    if arbre is None:
        return False
    if valeur < arbre.valeur:
        return rechercher(arbre.gauche,valeur)
    elif valeur > arbre.valeur:
        return rechercher(arbre.droit,valeur)
    else:
        return True
```

La recherche et l'ajout ont une complexité logarithmique en la taille de l'arbre ce qui est mieux que pour une structure de données linéaire

2 Encapsulation

On va créer une classe `ABR` avec un seul attribut `racine` initialisé à None (arbre vide)

```
class ABR:

    def __init__(self):
        self.racine = None
```

Ensuite on ajoute les méthodes `ajouter(valeur)` et `rechercher(valeu` utilisant les fonctions `ajouter(arbre,valeur)` et `rechercher(arbre,val` vues précédemment , ces dernières pouvant être vues comme des méthodes de classe

Pour Python les fonctions et les méthodes sont différentes bien qu'elles portent le même nom

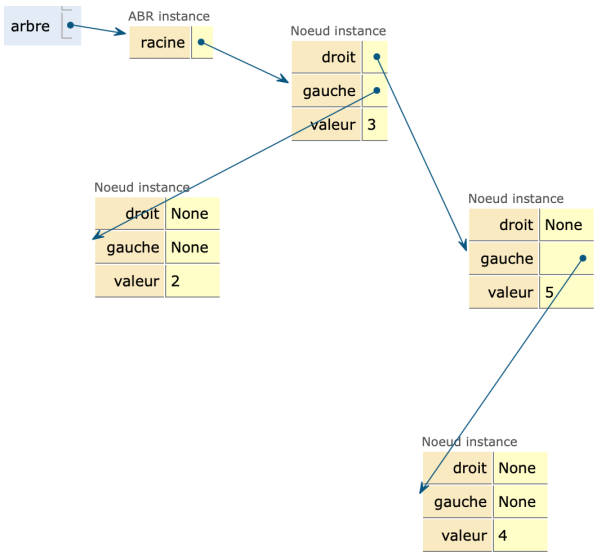
```
def ajouter(self,valeur):
    self.racine = ajouter(self.racine,valeur)

def rechercher(self,valeur):
    return rechercher(self.racine,valeur)
```

A l'exécution, par exemple

```
arbre = ABR()
arbre.ajouter(3)
arbre.ajouter(2)
arbre.ajouter(5)
arbre.ajouter(4)
```

On obtient l'arbre binaire de recherche suivant



3 Supprimer une valeur dans un ABR

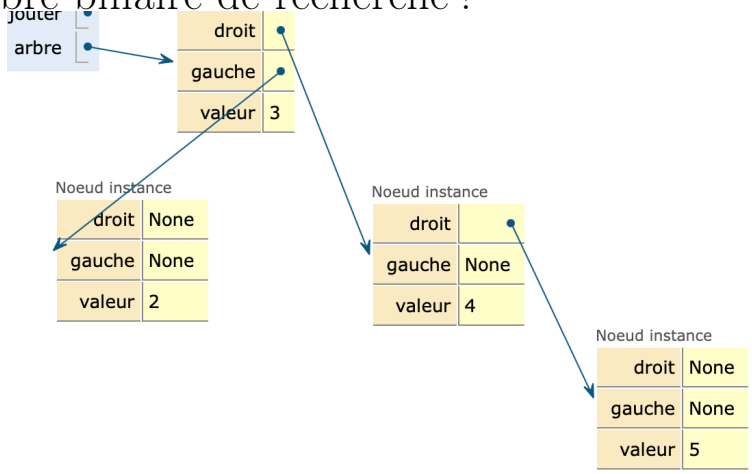
Comment supprimer une valeur dans un arbre binaire de recherche ?

Cette partie n'est pas au programme mais peut constituer l'objet d'une réflexion en exercice et/ou en TP

4 Exercices

Ex 1

Quelle est la suite d'instructions permettant d'obtenir cet arbre binaire de recherche ?



Ex 2

Dessiner l'arbre binaire de recherche obtenu après la suite d'instructions

```
arbre = ajouter(None,3)
arbre = ajouter(arbre,5)
arbre = ajouter(arbre,2)
arbre = ajouter(arbre,4)
```

Ex 3

Implémenter une fonction **itérative** `ajouter(arbre,valeur)`

Ex 4

Implémenter une méthode de la classe `ABR`, `valeur_max()` qui retourne la valeur maximale contenue dans l'arbre binaire de recherche

Défi

Proposer **votre** solution personnelle même imparfaite à la suppression d'une valeur dans un arbre binaire de recherche (Ne regarder ni les livres ni le Web)

BAC

1. Ex 3 Métropole Sujet 0 2021
2. Ex 3 Polynésie 2021