Arbre binaire de recherche

Le type `dict` (dictionnaire) de Python permet d'associer à une **clé** une **valeur**. On parle aussi de **tableau associatif** (par exemple dans le langage PHP) ou de **table de symboles**.

On définit de manière abstraite, la structure de données **table de symboles** (symbol table en anglais) permettant d'associer à une clé une valeur (en quelque sorte on généralise la structure de données, tableau).

Quelle est l'interface d'une table de symboles ?

1. `ajouter(tableau t,clé c,valeur v)` :
   ajouter dans le tableau t la clé c associée avec la valeur v (en Python `t[c] = v`)
2. `obtenir(tableau t,clé c)` :
   obtenir la valeur associée à la clé c (en Python `t[c]`)
3. `rechercher(tableau t,clé c)` :
   est ce que la clé c est dans le tableau t ?
   (en Python `c in t`)
4. `modifier(tableau t,clé c,valeur v)` :
   dans le tableau t à la clé associer la nouvelle valeur v
   (en Python `t[c] = v`)
5. `supprimer(tableau t,clé c)` :
   supprimer dans le tableau t la clé c (en Python `del t[c]`)

On suppose de plus que les **clés sont comparables** pour :

1. améliorer en terme de coût l'implémentation des fonctions de l'interface.
2. proposer des fonctions supplémentaires, comme `minimum(tableau t)` qui renvoie la plus petite clé du tableau t

Quelles sont les **implémentations** possibles d'une table de symboles ?

1. Avec un **arbre binaire de recherche**.
2. Avec une **table de hachage** (implémentation du dictionnaire de Python).

1 Arbre binaire de recherche

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

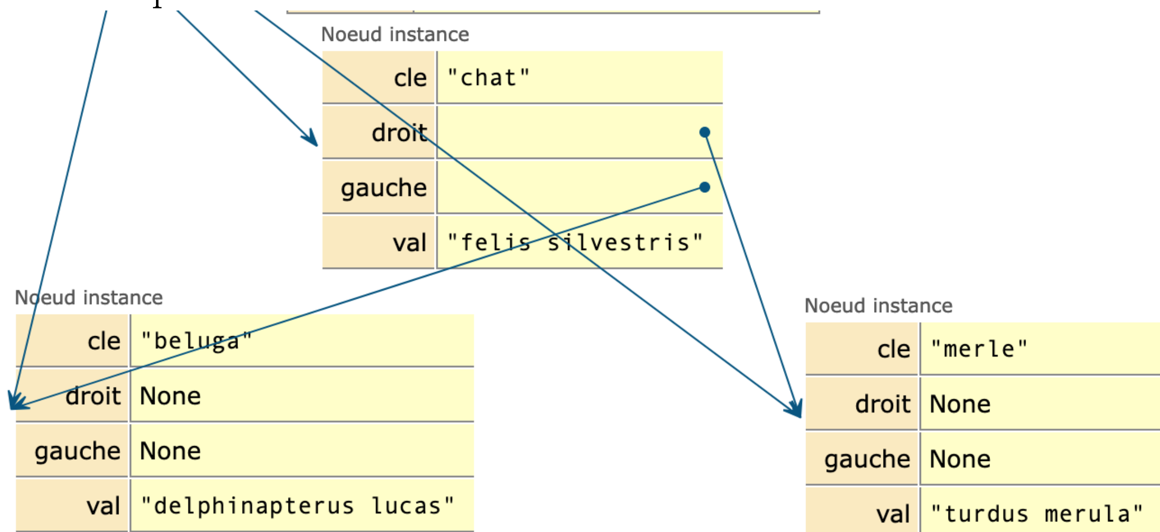
Toutes les clés des noeuds sont du même type et comparables, de plus

Pout tout noeud de clé n de l'arbre

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

Exemple :

On a visualisé avec Python Tutor un arbre binaire de recherche, où on associe à chaque nom d'animal le nom scientifique correspondant en latin.



obtenu à partir du code suivant

```
class Noeud:
    def __init__(self,c,v,g=None,d=None):
        self.cle = c
        self.val = v
        self.gauche = g
        self.droit = d

n1 = Noeud('beluga','delphinapterus lucas')
n2 = Noeud('merle','turdus merula')
n3 = Noeud('chat','felis silvestris',n1,n2)
```

On va implémenter une fonction récursive `ajouter(arbre,cle,val)` qui renvoie un arbre binaire de recherche dans lequel est ajouté un noeud contenant la clé `cle` et la valeur `val` . Si la clé est déjà dans l'arbre on renvoie l'arbre sans faire d'ajout.

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

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

```
arbre = ajouter(None,'chat','felis silvestris')
arbre = ajouter(arbre,'beluga',
```

```
'delphinapterus lucas')
arbre = ajouter(arbre,'merle','turdus merula')
```

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.
2. Il n'y a pas de doublons dans l'arbre binaire de recherche.

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

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

Comment obtenir la valeur associée à une clé donnée ?

```
def  obtenir(arbre,cle):
        if arbre is None:
            raise KeyError
        if cle < arbre.cle:
            return obtenir(arbre.gauche, cle)
        elif cle > arbre.cle:
            return obtenir(arbre.droit, cle)
        else:
            return arbre.val
```

Comment modifier la valeur associée à une clé ?
Si la clé ne se trouve pas dans l'arbre, on insère la clé et la valeur dans l'arbre.

```
def modifier(arbre, cle, val):
        if arbre is None:
            return Noeud(cle, val)
        if cle < arbre.cle:
            arbre.gauche = modifier(arbre.gauche,
            cle, val)
        elif cle > arbre.cle:
            arbre.droit = modifier(arbre.droit,
            cle, val)
        else:
            arbre.val = val
        return arbre
```

On verra plus loin le problème de la suppression d'une clé dans un arbre binaire de recherche.

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
```

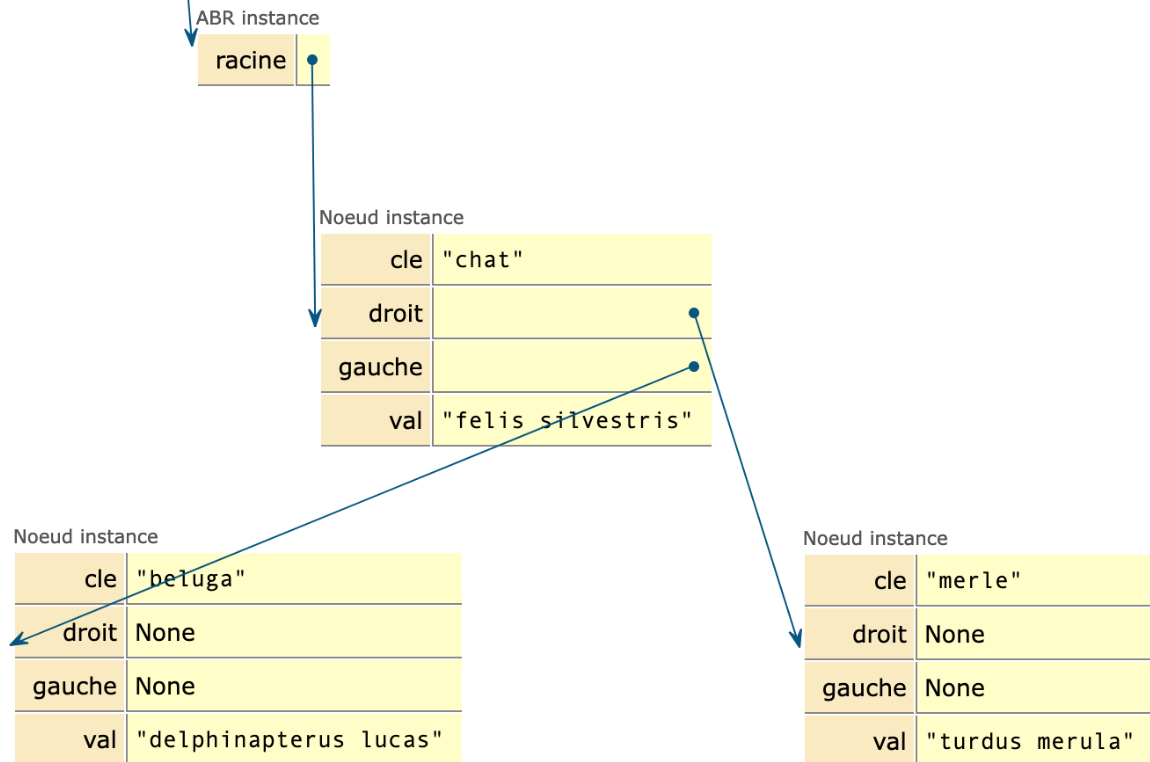
Ensuite on ajoute les méthodes correspondantes aux fonctions précédentes, **ces dernières pouvant être vues comme des méthodes de classe.**

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

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

    def ajouter(self,cle,val):
        self.racine = ajouter(self.racine,cle,val)

    def rechercher(self,cle):
        return rechercher(self.racine,cle)

    def obtenir(self,cle):
        return obtenir(self.racine,cle)

    def modifier(self,cle,val):
        return modifier(self.racine,cle,val)

arbre = ABR()
arbre.ajouter('chat','felis silvestris')
arbre.ajouter('beluga','delphinapterus lucas')
```

```
arbre.ajouter('merle','turdus merula')
```

On obtient l'arbre binaire de recherche suivant



3 Coût des fonctions et des méthodes

On a vu en exercices que la "forme" de l'arbre dépend de l'ordre dans laquelle on a inséré les clés au départ.

On peut donc avoir dans le meilleur des cas un arbre **parfait** et dans le pire une liste chaînée.

Dans le meilleur des cas la recherche d'une clé ou l'insertion d'un couple clé valeur est proportionnelle à la **hauteur** de l'arbre donc **proportionnelle au logarithme de la taille de l'arbre**.

Dans le pire des cas, la recherche d'une clé ou l'insertion d'un couple clé valeur est proportionnelle à la **taille** de l'arbre, ce qui n'est pas intéressant car on ne fait pas mieux que la structure de données, tableau ou liste chaînée.

Un autre problème se pose :

Que se passe-t-il au fur et à mesure des ajouts et des suppressions dans un arbre binaire de recherche même parfait au démarrage ?

Au bout d'un moment il risque d'évoluer vers une forme "linéaire" et les performances deviendront moins bonnes.

Pour résoudre ce problème on a développé à la fin du XXe siècle des arbres de recherche "équilibrés" qui conservent une forme arborescente avec :

$h \leqslant C \ln(n)$ où h est la hauteur de l'arbre n la taille et C une constante

Ainsi les opérations d'ajout de recherche et de suppression ont un coût proportionnel à la hauteur de l'arbre et donc un coût logarithmique en la taille de l'arbre.

4 Supprimer la clé maximale dans un ABR

La clé maximale dans un ABR (sans doublons) n'a pas de fils droit (voir exercice) et éventuellement un sous arbre gauche.

Supprimer la clé maximale d'un ABR consiste à raccorder le sous arbre gauche de la clé maximale en tant que sous arbre droit de l'ancêtre de la clé maximale.

Ce qui peut être traduit en itératif mais est plus joliment traduit en récursif de la façon suivante :

```
def supprime_max(a:Noeud)->Noeud:
    """
    a est de type Noeud non vide
    """
    if a.droit is None:
        return a.gauche
    return Noeud(a.cle,a.val,
```

```
    a.gauche,supprime_max(a.droit))
```

Exercices

Ex 1

1. Donner tous les arbres binaires de recherche dont les noeuds contiennent les valeurs 1, 2 et 3
2. Donner tous les arbres binaires de recherche dont les noeuds contiennent les valeurs 'a', 'b', 'c' et 'd'.

Ex 2

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

```
arbre = ajouter(None,'a','alpha')
arbre = ajouter(arbre,'b','beta')
arbre = ajouter(arbre,'r','rho')
```

Ex 3

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

```
arbre = ABR()
arbre = arbre.ajouter('b','beta')
arbre = arbre.ajouter(,'a','alpha')
arbre = arbre.ajouter(,'r','rho')
```

Ex 4

1. Créer une classe `ABR2` de la manière suivante sans utiliser de fonctions externes :

```
class ABR2:
    def __init__(self):
        self.root = None
```

```
    def put(self,p,key,val):
        """
        p est une référence ou pointeur
        sur un sous-arbre de self
        """
        pass

    def contains(self,p,key):
        pass

    def get(self,p,key):
        pass

    def set(self,p,key,val):
        pass
```

2. On va réécrire la classe (classe ABR3) afin de pouvoir **utiliser l'écriture de Python pour les dictionnaires** :
   (a) Pour *ajouter un couple (clé, valeur) ou modifier une valeur* on écrit `t[cle] = valeur`
   (b) Pour obtenir la valeur associée à une clé on écrit `t[cle]`
   (c) Pour savoir si uné clé est présente dans le tableau on écrit `cle in t`

```
class ABR3:
    def __init__(self):
        self.root = None

   def _get(self, p, key):
        """
```

```
        méthode privée, c'est la méthode get de
        """
        pass

    def __getitem__(self, key):
        """
        méthode spéciale de Python permettant
        d'écrire t[key]
        """
        return self._get(self.root, key)

    def _set(self, p, key, val):
        pass

    def __setitem__(self, key, val):
        """
        méthode spéciale de Python permettant
        d'écrire t[key] = val
        """
        self.root = self._set(self.root, key, va

    def _contains(self, p, key):
        pass

    def __contains__(self, key):
        return self._contains(self.root, key)
```

Ex 5

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

Ex 6

1. Justifier que la plus grande clé dans un ABR n'a pas de

fils droit (faire un raisonnement par l'absurde)

2. Donner un exemple d'ABR où la plus grande clé ne se trouve pas pas une feuille de l'arbre.
3. Implémenter une méthode **récursive** de la classe `ABR`, `maximum()` qui retourne la clé maximale contenue dans l'arbre binaire de recherche.

Ex 6

1. Justifier que le parcours infixe d'un ABR affiche les clés dans l'ordre croissant.
2. D'où l'idée d'écrire une fonction Python `tri_ABR(t:list)` qui prend en paramètre un tableau et qui renvoie un nouveau tableau trié dans l'ordre croissant (Première phase :Parcourir chaque élément du tableau et l'insérer dans un arbre binaire de recherche ; deuxième phase :faire le parcours infixe de cet arbre)
3. Evaluer la complexité de la première phase, puis celle de la deuxième phase sur deux cas particuliers
   (a) Le tableau est dans l'ordre décroissant.
   (b) Le tableau est de telle sorte que l'arbre binaire de recherche créé est parfait.
4. En conclusion quelle est la complexité de cet façon de procéder dans le pire des cas ?

Ex 7

Définir une méthode récursive `supprime_minimum()` qui retourne le même arbre sans le minimum de l'arbre.

Ex 8

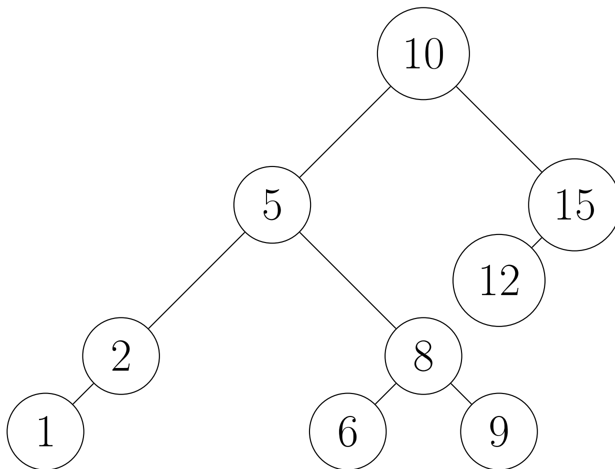
Ecrire une fonction récursive `supprime(c,a)` qui prend en paramètres une clé `c` et un arbre binaire de recherche `a` de type

`Noeud` et qui renvoie un arbre binaire de recherche dans lequel `c` a été supprimé.

Indications : Si la clé supprimée est à la racine de l'arbre renvoyer un abr dont la nouvelle racine est le maximum du sous arbre gauche, le sous arbre gauche étant le sous arbre gauche à qui on a supprimé le maximum, le sous arbre droit reste inchangé. On utilisera les fonctions `maximum` et `supprime_maximum`

Ex 9

On définit le **successeur** d'un noeud x dans un ABR (sans doublons) comme étant le noeud y de telle sorte que $x.cle$ et $y.cle$ se succèdent dans l'ordre croissant des valeurs de l'ABR



1. Donner le successeur de :
   (a) 10
   (b) 5
   (c) 2
   (d) 9
2. Justifier : si x a un fils droit alors le successeur de x est dans le sous arbre droit de x le plus à gauche possible (descente)
3. Justifier : si x n'a pas de fils droit alors le successeur y de x. est l'ancêtre le plus "jeune" de x dont l'enfant de gauche est aussi un ancêtre de x (montée)

4. Afin de définir une méthode `successeur(x)` qui retourne l'ABR dont la racine est le successeur du noeud x, on ajoute un attribut `pere` à la classe `Noeud` qui vaut `None` si le noeud est la racine de l'ABR et qui est la référence de l'ancêtre du noeud sinon

   Une fois cette modification faite définir `successeur(x)`

5. Comment utiliser `successeur(x)` pour supprimer x dans l'arbre binaire de recherche ?

Ex 10

Un arbre binaire est dit dans cet exercice "équilibré" si ses deux sous-arbres gauche et droit le sont et si les hauteurs de ses deux sous arbres diffèrent au plus de 1.

Implémenter une méthode `est_equilibre()` de coût linéaire qui renvoie Vrai si l'arbre binaire est équilibré et Faux sinon.

BAC

1. Ex 3 Polynésie 2 2024
2. Ex 3 Métropole Sujet 0 2021
3. Ex 3 Polynésie 2021