TP6

May 24, 2016

1 Arbre binaire de Recherche

Le but de ce TP est de manipuler des Arbres Binaires de Recherche. Rappel : ce sont des arbres binaires dont les noeuds contiennent des valeurs telles que pour chaque noeud n de valeur x : * les valeurs du sous arbres gauche de n sont inférieures ou égales à x * les valeurs du sous arbre droit de n sont strictement supérieures à x

1.0.1 Rappel: Papier et crayon sont des outils souvent indispensables à l'informaticien·ne.

1.1 Structure

On utilisera la classe suivante qui donne une implatantion minimale des arbres binaires.

```
In [3]: class ArbreBinaire():

            def __init__(self, x, filsG = None, filsD = None): # constructeur
                self.valeur = x
                self.filsG = filsG
                self.filsD = filsD

            def __repr__(self): # affichage
                if self.filsG is None and self.filsD is None:
                    return str(self.valeur)
                s = str(self.valeur) + "[" + str(self.filsG) + ", " + str(self.filsD) + "]"
                s = s.replace("None",".") # pour alléger l'affichage
                return s
```

Voilà par exemple la création d'un arbre binaire de racine 3, avec un fils gauche de valeur 1 et un fils droit de valeur 4.

```
In [4]: N1 = ArbreBinaire(1)
        N4 = ArbreBinaire(4)
        N3 = ArbreBinaire(3,N1,N4)
        N3

Out[4]: 3[1, 4]
```

En voici un autre, cette fois, 4 a aussi un fil droit de valeur 5.

```
In [5]: N1 = ArbreBinaire(1)
        N5 = ArbreBinaire(5)
        N4 = ArbreBinaire(4,None,N5)
        N3 = ArbreBinaire(3,N1,N4)
        N3
```

```
Out[5]: 3[1, 4[., 5]]
```

On peut accéder aux différents champs de cette façon :

```
In [6]: print N3.valeur
        print N3.filsG
        print N3.filsD
```

```
3
1
4[., 5]
```

Exercice : créer l'ABR où l'on a rajouté 2 comme fils droit du sous-arbre enraciné en 1 de l'exemple précédent.

```
In [1]: # écrire le code ici
```

1.2 Insertion

On rappelle l'algorithme d'insertion dans un arbre binaire de recherche : * l'insertion dans un noeud vide renvoie l'arbre contenant le noeud à insérer, * sinon, on compare la valeur à insérer à la valeur du noeud, on insère à gauche si la valeur est plus petite ou égale et à droite sinon

```
In [2]: """
        Insertion dans un ABR
        Input :
            - A, l'arbre binaire de recherche (peut être None)
            - v, la valeur à insérer
        Output :
            la nouvelle racine de l'arbre
        """
        def insertABR(A,v):
            # écrire le code ici
            pass
```

```
In [11]: A = insertABR(None,3)
         assert(str(A) == "3")
         A = insertABR(A,1)
         assert(str(A) == "3[1, .]")
         A = insertABR(A,4)
         assert(str(A) == "3[1, 4]")
         A = insertABR(A,5)
         assert(str(A) == "3[1, 4[., 5]]")
         A = insertABR(A,2)
         assert(str(A) == "3[1[., 2], 4[., 5]]")
         A = insertABR(A,1)
         assert(str(A) == "3[1[1, 2], 4[., 5]]")
```

```
In [ ]: """
        Insertion d'une liste dans un ABR

        Input :
            - L, une liste de valeurs
        Output :
            L'arbre binaire obtenu en insérant les éléments un à un
        """
```

```
        def insertListABR(L):
            # écrire le code ici
            pass
```

```
In [ ]: A = insertListABR([3,1,4,5,2,1])
        assert(str(A) == "3[1[1, 2], 4[., 5]]")
        A = insertListABR([3,4,5,1,2,1])
        assert(str(A) == "3[1[1, 2], 4[., 5]]")
        A = insertListABR([5,1,4,2,5,8,5,6,7,2,5])
        assert(str(A) == "5[1[., 4[2[2, .], 5[5[5, .], .]]], 8[6[., 7], .]]")
```

On vous donne la fonction suivante qui permet de récupérer les éléments de l'arbre en ordre infixe et de les mettre dans une liste.

Techniquement on utilise le fantastique système de générateurs en Python. On utilise l'instruction yield qui permet d'énumérer l'ensemble à générer. L'instruction yield agit un peu comme un return qui n'arête pas l'exécution du programme, mais le met en pause jusqu'à son prochain appel.

```
In [ ]: def infixeGenerator(A):
            if A is None:
                return
            for v in infixeGenerator(A.filsG):
                yield v
            yield A.valeur
            for v in infixeGenerator(A.filsD):
                yield v

        def infixeList(A):
            return list(infixeGenerator(A))
```

On peut maintenant vérifier que le tri par ABR... trie bien les valeurs comme escompté!

```
In [ ]: A = insertListABR([3,1,4,5,2,1])
        assert(infixeList(A) == [1,1,2,3,4,5])
        A = insertListABR([3,4,5,1,2,1])
        assert(infixeList(A) == [1,1,2,3,4,5])
        A = insertListABR([5,1,4,2,5,8,5,6,7,2,5])
        assert(infixeList(A) == [1,2,2,4,5,5,5,5,6,7,8])
```

1.3 Rotations

Reprenez les algorithmes (vus en TD) de rotation droite et rotation gauche et implantez les.

```
In [ ]: """
        Rotation droite
        Effectue une rotation droite à la racine de l'arbre donné en argument.
        Si l'arbre est vide ou si son sous-arbre gauche est vide, la rotation ne fait rien.

        Input :
            - T un ABR
        Output:
            La nouvelle racine de l'arbre
        """
        def rightRotation(T):
            # écrire le code ici
            pass
```

In [ ]:
```
T = None
assert(rightRotation(T) == T)
T = ArbreBinaire(2)
assert(rightRotation(T) == T)
T = ArbreBinaire(2,None, ArbreBinaire(3))
assert(rightRotation(T) == T)
T = ArbreBinaire(2, ArbreBinaire(1), None)
T = rightRotation(T)
assert(str(T) == "1[., 2]")
T = insertListABR([5,7,3,4,1,6,2])
assert(str(T) == "5[3[1[., 2], 4], 7[6, .]]")
T = rightRotation(T)
assert(str(T) == "3[1[., 2], 5[4, 7[6, .]]]")
T = insertListABR([4,5,7,3,2,1,6])
assert(str(T) == "4[3[2[1, .], .], 5[., 7[6, .]]]")
T = rightRotation(T)
assert(str(T) == "3[2[1, .], 4[., 5[., 7[6, .]]]]")
T = insertListABR([5,2,7,4,3,6,1])
assert(str(T) == "5[2[1, 4[3, .]], 7[6, .]]")
T = rightRotation(T)
assert(str(T) == "2[1, 5[4[3, .], 7[6, .]]]")
```

In [ ]:
```
"""
Rotation gauche
Effectue une rotation gauche à la racine de l'arbre donné en argument.
Si l'arbre est vide ou si son sous-arbre droit est vide, la rotation ne fait rien.

Input :
    - T un ABR
Output:
    La nouvelle racine de l'arbre
"""
def leftRotation(T):
    # écrire le code ici
    pass
```

In [ ]:
```
T = None
assert(leftRotation(T) == T)
T = ArbreBinaire(2)
assert(leftRotation(T) == T)
T = ArbreBinaire(2,ArbreBinaire(1), None)
assert(leftRotation(T) == T)
T = ArbreBinaire(2, None, ArbreBinaire(3))
T = leftRotation(T)
assert(str(T) == "3[2, .]")
T = insertListABR([3,1,5,2,4,7,6])
assert(str(T) == "3[1[., 2], 5[4, 7[6, .]]]")
T = leftRotation(T)
assert(str(T) == "5[3[1[., 2], 4], 7[6, .]]")
T = insertListABR([3,4,5,2,7,6,1])
assert(str(T) == "3[2[1, .], 4[., 5[., 7[6, .]]]]")
T = leftRotation(T)
assert(str(T) == "4[3[2[1, .], .], 5[., 7[6, .]]]")
T = insertListABR([2,5,7,1,6,4,3])
assert(str(T) == "2[1, 5[4[3, .], 7[6, .]]]")
```

```
T = leftRotation(T)
assert(str(T) == "5[2[1, 4[3, .]], 7[6, .]]")
```

1.4 Equilibre

On dit qu'un ABR est équilibré si la différence de hauteur à gauche et à droite de tous les noeuds est strictement ingérieure à 2 (donc -1, 0 ou 1).

Le but de cette section est d'écrire un algorithme d'insertion dans les arbres équilibrés, qui préserve cet propriété. Pour cela, on va sauvegarder au niveau de chaque noeud la hauteur du sous-arbre issu du noeud. Cela permetra de connaitre en temps constant l'état d'équilibre d'un arbre.

Dans la littérature, on appelle AVL cette classe d'ABR. Le nom AVL provient du nom des deux auteurs de l'algorithme: Georgii Adelson-Velsky et Evguenii Landis.

On crée une nouvelle classe AVL pour les arbres avec paramètres de hauteur.

In [ ]:
```
class AVL():

    def __init__(self, x): # constructeur
        self.valeur = x
        self.hauteur = 1
        self.filsG = None #  On ne permet plus l'initialisation de filsG et filsD
        self.filsD = None #  ils ne seront modifés que par l'insertion

    def __repr__(self): # affichage
        if self.filsG is None and self.filsD is None:
            return str(self.valeur)
        s = str(self.valeur) + "[" + str(self.filsG) + ", " + str(self.filsD) + "]"
        s = s.replace("None",".") # pour alléger l'affichage
        return s

    def equilibre(self):
        """
        Retourne la valeur d'équilibre de l'arbre en fonction des hauteurs sauvegardées
        """
        hg = 0
        if not self.filsG is None:
            hg = self.filsG.hauteur
        hd = 0
        if not self.filsD is None:
            hd = self.filsD.hauteur
        return hg - hd

    # la méthode suivante va nous permettre de vérifier que le paramètre de hauteur est mis cor

    def checkHauteur(self):
        """
        Calcule récursivement la hauteur et vérifie la cohérence des valeurs sauvegardées

        Output:
            - True ou False selon que les valeurs de hauteur correspondent à la vrai hauteur de
            - la hauteur de l'arbre calculée par l'algorithme
        """
        if not self.filsG is None:
            cg, hg = self.filsG.checkHauteur()
        else:
```

```
            cg, hg = True, 0 # fils vide
        if not self.filsD is None:
            cd, hd = self.filsD.checkHauteur()
        else:
            cd, hd = True, 0 # fils vide
        h = max(hg,hd) +1
        if not cg or not cd: # les valeurs sont incohérentes sur les fils
            return False, h
        if not self.hauteur == h:
            return False, h
        return True, h

    def checkAVL(self):
        """
        Vérifie que l'arbre est un AVL en utilisant les valeurs des hauteurs stockées.
        """
        return self.equilibre() > -2 and self.equilibre() < 2 and (self.filsG is None or self.fi
```

Il faut maintenant modifier légèrement notre algorithme d'insertion pour prendre en compte la mise à jour du paramètre de hauteur. On effectue une insertion classique puis on calcule le nouveau paramètre en fonction de celui des fils.

In [ ]:
```
"""
Insertion avec mise à jour de la hauteur

Input :
    - T, un arbre binaire de la classe AVL
    - v, la valeur à insérer
"""
def insertHauteur(T, v):
    # écrire le code ici
    pass
```

In [ ]:
```
A = insertHauteur(None,3)
assert(str(A) == "3")
assert(A.checkHauteur()[0])
A = insertHauteur(A,1)
assert(str(A) == "3[1, .]")
assert(A.checkHauteur()[0])
A = insertHauteur(A,4)
assert(str(A) == "3[1, 4]")
assert(A.checkHauteur()[0])
A = insertHauteur(A,5)
assert(str(A) == "3[1, 4[., 5]]")
assert(A.checkHauteur()[0])
A = insertHauteur(A,2)
assert(str(A) == "3[1[., 2], 4[., 5]]")
assert(A.checkHauteur()[0])
A = insertHauteur(A,1)
assert(str(A) == "3[1[1, 2], 4[., 5]]")
assert(A.checkHauteur()[0])
```

In [ ]:

In [ ]:
```
"""
Insertion d'une liste dans un Arbre binaire avec mise à jour des hauteurs
```

```
        Input :
            - L, une liste de valeur
        Output :
            L'arbre binaire obtenu de classe AVL
        """
        def insertListHauteur(L):
            # écrire le code ici
            pass
```

```
In [ ]: A = insertListHauteur([3,1,4,5,2,1])
        assert(infixeList(A) == [1,1,2,3,4,5])
        assert(A.checkHauteur()[0])
        A = insertListHauteur([3,4,5,1,2,1])
        assert(infixeList(A) == [1,1,2,3,4,5])
        assert(A.checkHauteur()[0])
        A = insertListHauteur([5,1,4,2,5,8,5,6,7,2,5])
        assert(infixeList(A) == [1,2,2,4,5,5,5,5,6,7,8])
        assert(A.checkHauteur()[0])
```

On va avoir besoin des fonctions de rotation, cependant, il faut faire attention à ce que les rotation modifient les hauteurs des différents noeuds. On écrit donc une nouvelle fonction qui, après l'opération de rotation, met à jour les hauteurs de x et de y.

```
In [ ]: """
        Rotation droite avec mise à jour des hauteurs
        Effectue une rotation droite à la racine de l'arbre donné en argument et met à jour les valeurs

        Input :
            - T un AVL
        Output:
            La nouvelle racine de l'arbre
        """
        def rightRotationHauteur(T):
            # écrire le code ici
            pass
```

```
In [ ]: T = insertListHauteur([5,7,3,4,1,6,2])
        assert(str(T) == "5[3[1[., 2], 4], 7[6, .]]")
        T = rightRotationHauteur(T)
        assert(str(T) == "3[1[., 2], 5[4, 7[6, .]]]")
        assert(T.checkHauteur()[0])
        T = insertListHauteur([4,5,7,3,2,1,6])
        assert(str(T) == "4[3[2[1, .], .], 5[., 7[6, .]]]")
        T = rightRotationHauteur(T)
        assert(str(T) == "3[2[1, .], 4[., 5[., 7[6, .]]]]")
        assert(T.checkHauteur()[0])
        T = insertListHauteur([5,2,7,4,3,6,1])
        assert(str(T) == "5[2[1, 4[3, .]], 7[6, .]]")
        T = rightRotationHauteur(T)
        assert(str(T) == "2[1, 5[4[3, .], 7[6, .]]]")
        assert(T.checkHauteur()[0])
```

```
In [ ]: """
        Rotation gauche avec mise à jour des hauteurs
```

```
        Effectue une rotation gauche à la racine de l'arbre donné en argument et met à jour les valeurs
        Si l'arbre est vide ou si son sous-arbre droit est vide, la rotation ne fait rien.

        Input :
            - T un AVL
        Output:
            La nouvelle racine de l'arbre
        """
        def leftRotationHauteur(T):
            # écrire le code ici
            pass
```

```
In [ ]: T = insertListHauteur([3,1,5,2,4,7,6])
        assert(str(T) == "3[1[., 2], 5[4, 7[6, .]]]")
        T = leftRotationHauteur(T)
        assert(str(T) == "5[3[1[., 2], 4], 7[6, .]]")
        assert(T.checkHauteur()[0])
        T = insertListHauteur([3,4,5,2,7,6,1])
        assert(str(T) == "3[2[1, .], 4[., 5[., 7[6, .]]]]")
        T = leftRotationHauteur(T)
        assert(str(T) == "4[3[2[1, .], .], 5[., 7[6, .]]]")
        assert(T.checkHauteur()[0])
        T = insertListHauteur([2,5,7,1,6,4,3])
        assert(str(T) == "2[1, 5[4[3, .], 7[6, .]]]")
        T = leftRotationHauteur(T)
        assert(str(T) == "5[2[1, 4[3, .]], 7[6, .]]")
        assert(T.checkHauteur()[0])
```

On a maintenant tous les outils pour écrire une vraie insertion AVL, c'est-à-dire, une insertion qui conserve des arbres équilibrés. Le principe est le suivant : on suppose que l'on est sur un noeud n et l'insertion a modifié sa valeur d'équilibre pour l'amener à 2 ou -2, on supose que les sous-arbres ont déjà été rééquilibrés. Plusieurs cas sont possibles : * l'équilibre de n est de 2 et l'équilibre de son fils gauche est de 1 : **on applique une rotation droite sur** n * l'équilibre de n est de 2 et l'équilibre de son fils gauche est de -1 : **on applique une rotation gauche sur le fils gauche puis une rotation droite sur** n * l'équilibre de n est de -2 et l'équilibre de son fils droit est de -1 : **on applique une rotation gauche sur** n * l'équilibre de n est de -2 et l'équilibre de son fils droit est de 1 : **on applique une rotation droite sur le fils droit puis une rotation gauche sur** n.

Il est possible de montrer que ce sont les **seules possibilités**, vous pouvez essayer de voir vous-même pourquoi (un excellent exercice). Remarquez que lorsqu'on insère, les sous-arbres sont rééquilibrés avant la racine. Donc l'algorithme d'insertion dans un arbre non vide est le suivant : * on insère à gauche ou à droite en fonction de la valeur à insérer, * on met à jour la valeur de hauteur du noeud * si l'arbre obtenu est déséquilibré, on le rééquilibre avec l'algorithme décrit ci-dessus.

```
In [ ]: """
        Insertion AVL
        insertion d'une valeur dans un AVL avec rééquilibrage

        Input :
            - T, un arbre binaire de la classe AVL
            - v, la valeur à insérer
        """
        def insertAVL(T, v):
            # écrire le code ici
            pass
```

In [ ]:
```
"""
Insertion d'une liste dans un AVL

Input :
    - L, une liste de valeur
Output :
    L'AVL obtenu
"""
def insertListAVL(L):
    # écrire le code ici
    pass
```

In [ ]:
```
A = insertListAVL([3,1,4,5,2])
assert(infixeList(A) == [1,2,3,4,5])
assert(A.checkHauteur()[0])
assert(A.checkAVL())
A = insertListAVL([3,4,5,1,2])
assert(infixeList(A) == [1,2,3,4,5])
assert(A.checkHauteur()[0])
assert(A.checkAVL())
A = insertListAVL([1,2,3,4,5,6,7,8,9,10,11])
assert(infixeList(A) == [1,2,3,4,5,6,7,8,9,10,11])
assert(A.checkHauteur()[0])
assert(A.checkAVL())
A = insertListAVL([11,10,9,8,7,6,5,4,3,2,1])
assert(infixeList(A) == [1,2,3,4,5,6,7,8,9,10,11])
assert(A.checkHauteur()[0])
assert(A.checkAVL())
A = insertListAVL([11,1,10,2,9,3,8,4,7,5,6])
assert(infixeList(A) == [1,2,3,4,5,6,7,8,9,10,11])
assert(A.checkHauteur()[0])
assert(A.checkAVL())
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: