Equilibrage des arbres

1. Arbres déséquilibrés ?

Création d'arbres déséquilibrés

Enoncé

1. Créer quatre ABR `tree1`, `tree2`, ..., à partir des listes suivante :

```
l1 = [5, 2, 11, 3, 7, 13, 9]
l2 = [7, 3, 11, 2, 5, 9, 13]
l3 = [2, 3, 5, 11, 7, 13, 9]
l4 = [2, 3, 5, 7, 9, 11, 13]
```

2. Que peut-on dire des quatre listes précédentes.
3. Afficher les 4 arbres obtenus.
4. En considérant que le temps de recherche dans un ABR est directement proportionnel à la hauteur du noeud recherché, le temps moyen pour trouver dans `tree1` :
   - le noeud 5 est de 1 ;
   - les noeuds 2 et 11 de 2 ;
   - les noeuds 3, 7 et 13 de 3 ;
   - le noeud 9 de 4.

   le temps moyen de recherche sur cet arbre est donc $\frac{1+2\times 2+3\times 3+4}{7}=\frac{18}{7}$.

   Calculer de même le temps moyen pour les ABR `tree2`, `tree3` et `tree4`. Que peut-on en conclure ?
5. On considère l'arbre équilibré `tree2`. Insérer maintenant les éléments suivants dans l'arbre : 17, 14, 15, 16. Comment est l'arbre obtenu ?

Solution

A venir !

On comprend aisément qu'un arbre équilibré (solution 2) donne en moyenne de meilleurs résultats que tout autre arbre, et qu'un arbre dégénéré (solution 4) donne des résultats plus mauvais que toute autre représentation. La notion formelle **d'arbre équilibré** (*balanced tree* chez Donald Knuth) n'est pas au programme, mais elle peut être abordée intuitivement.

Sur un ABR équilibré, la recherche d'un élément est en moyenne en $log_2(n)$, comme avec une recherche par dichotomie dans une liste. On peut alors se poser la question de l'intérêt d'utiliser un ABR. La réponse tient dans le temps mis à ajouter / supprimer un élément : dans une liste, on est en $\mathcal{O}(n)$ (complexité linéaire), alors qu'on est en $\mathcal{O}(log_2(n))$ dans un ABR (complexité logarithmique).

Aussi, lorsqu'on doit stocker une collection d'éléments ordonnés, une liste Python peut être utilisée si la collection évolue peu ou bien si les éléments sont stockés dans l'ordre du tri. Mais si la collection doit être régulièrement modifiée et que les recherches par rapport au critère de tri sont fréquentes, on privilégie les ABR.

Il est donc particulièrement important que notre ABR soit équilibré, et surtout que les insertions/délétions conservent la propriété d'équilibrage de ces ABR. Il existe plusieurs techniques pertmettant d'obtenir des ABR équilibrés :

- les arbres AVL ;
- les arbres rouge-noir ;

- les arbres 2-3 ;
- les arbres 2-3-4 ;
- les B-arbres.

2. Les arbres AVL

2.1. Présentation des AVL

En informatique théorique, les arbres AVL ont été historiquement les premiers arbres binaires de recherche **automatiquement équilibrés**.

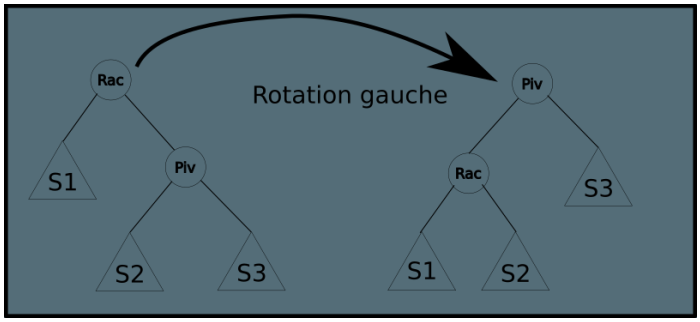
La dénomination « arbre AVL » provient des noms respectifs de ses deux inventeurs, respectivement Georgii Adelson-Velsky et Evguenii Landis, qui l'ont publié en 1962 sous le titre *An Algorithm for the Organization of Information*.

Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un, et cette propriété est conservée dynamiquement au moment de l'insertion ou de la déletion d'un nouveau noeud. La recherche, l'insertion et la suppression sont toutes en $\mathcal{O}(log_2(n))$ dans le pire des cas.

Cependant les mécanismes d'insertion et de déletions sont modifiés par rapport à ceux déjà travaillés : ils nécessitent des **rotations droites et gauches**.

2.2. Rotations droites et gauches

Le mécanisme de rotation gauche consiste à remplacer un noeud **racine** par son noeud droit (appelé **pivot**).



Après l'opération :

- les noeuds du sous-arbre `S1` demeurent bien inférieurs au noeud `Racine` ;
- les noeuds du sous-arbre `S2`, qui étaient tous supérieurs à `Racine` tout en étant inférieurs à `Pivot`, le demeurent toujours ;
- les noeuds du sous-arbre `S2` demeurent bien supérieurs au noeud `Pivot`.

Ainsi par une rotation gauche, le nouvel arbre obtenu est aussi un ABR.

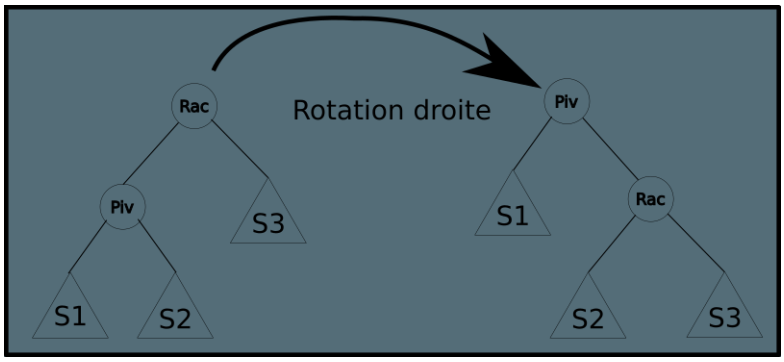
Une version algorithmique de cette implémentation est :

```
Pivot <- Racine.droit
Racine.droit <- Pivot.gauche
Pivot.gauche <- Racine
Racine <- Pivot
```

Implémentation

Enoncé

1. Créer une méthode `rotationGauche` pour la classe `Node`. Attention ! Pour pouvoir implémenter correctement cette méthode, **il ne faudra pas oublier de mettre à jour les parents de chaque noeud !**
2. Créer de mémé une méthode `rotationDroite` pour la classe `Node`, en s'aidant du schéma suivant :



Solution

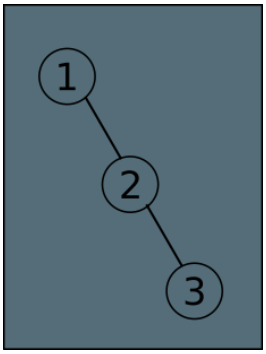
A venir !

2.3. Techniques d'ésuilibrages :

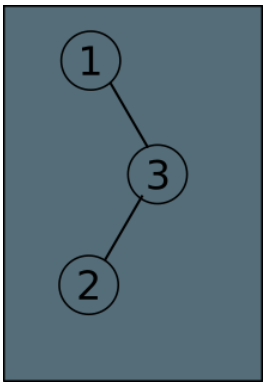
Application manuelle

Enoncé

1. Appliquer **à la main** une rotation gauche sur la racine de l'arbre suivant. Le résultat est-il plus équilibré ?



2. Appliquer **à la main** une rotation gauche sur la racine de l'arbre suivant. Le résultat est-il plus équilibré ?



3. Appliquer une rotation droite sur le noeud 3 de l'arbre précédent, puis une rotation gauche sur la racine de l'arbre obtenu. Que peut-on en conclure ?

Solution

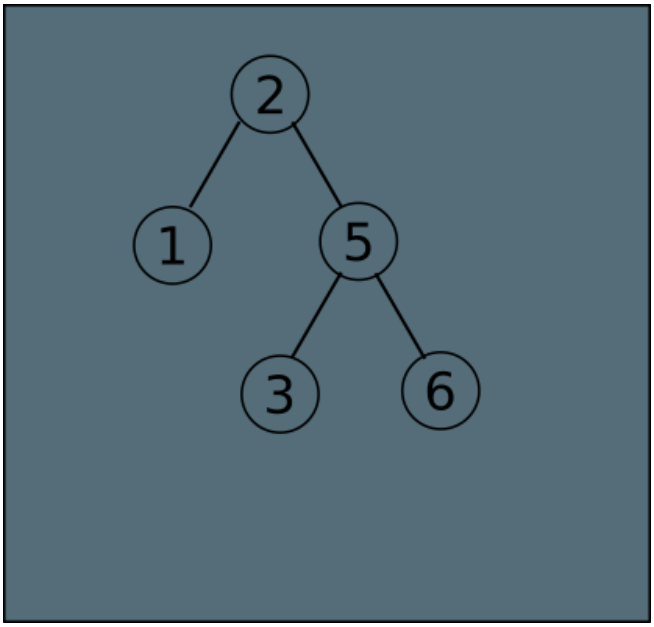
A venir !

En pratique on équilibrera un arbre à chaque insertion et à chaque délétion. Pour ce faire, une fois le noeud inséré, on regardera la hauteur de chaque sous-arbre gauche et droit, et si ces hauteurs diffèrent de plus de 1, on effectuera les rotations nécessaires.

Application manuelle 2

Enoncé

1. On considère l'abre suivant :



   Est-il équilibré au sens donné ?
2. On insère maintenant la valeur 4. L'arbre est-il équilibré ?
3. Appliquer une rotation gauche sur la racine. Cela suffit-il pour que le résultat soit équilibré ?
4. En repartant de l'arbre de la question 2, quelle roitation faudrait-il faire avant d'appliquer une rotation gauche sur la racine afin d'obtenir un arbre équilibré ?

Solution

A venir !

L'algorithme réel d'équilbrage étant un peu trop complexe pour notre niveau, voici les codes à insérer dans les différentes classes pour obtenir un AVL :

- dans la classe `ABR` :

```
def insererAVL(self, valeur):
    if self.estVide():
        self.racine = Node(valeur)
    else:
        self.racine = self.racine.inserer_AVL(valeur)
```

- dans la classe `Node` :

```
def inserer_AVL(self, valeur):
    if valeur < self.valeur:
        if self.gauche is None:
            self.gauche = Node(valeur, parent=self)
            return self
        else:
            self.gauche = self.gauche.insererAVL(valeur)
            return self.equilibrer()
```

```
    elif valeur > self.valeur:
        if self.droit is None:
            self.droit = Node(valeur, parent=self)
            return self
        else:
            self.droit = self.droit.insererAVL(valeur)
            return self.equilibrer()
    else:
        return self

def equilibrer(self):
    hauteur_gauche = hauteur(self.gauche)
    hauteur_droit = hauteur(self.droit)
    if hauteur_gauche - hauteur_droit == 2:
        hauteur_gauche_gauche = hauteur(self.gauche.gauche)
        hauteur_gauche_droit = hauteur(self.gauche.droit)
        if hauteur_gauche_gauche > hauteur_gauche_droit:
            return self.rotationDroite()
        else:
            self.gauche = self.gauche.rotationGauche()
            return self.rotationDroite()
    elif hauteur_gauche - hauteur_droit == -2:
        hauteur_droit_droit = hauteur(self.droit.droit)
        hauteur_droit_gauche = hauteur(self.droit.gauche)
        if hauteur_droit_droit > hauteur_droit_gauche:
            return self.rotationGauche()
        else:
            self.droit = self.droit.rotationDroite()
            return self.rotationGauche()
    else:
        return self

def hauteur(self):
    hauteur_gauche = hauteur(self.gauche)
    hauteur_droit = hauteur(self.droit)
    return 1 + max(hauteur_gauche, hauteur_droit)
```

Désormais, en créantun ABR et en utilisant la méthode `insertionAVL`, l'arbre obtenu doit être automatiquement équilibré.