Fiche de TP no 4

L'objectif du TP est donner une implémentation des dictionnaires en utilisant les arbres binaires de recherche. Rappelons la définition de la classe abstraite des dictionnaires :

```
class Dictionary():
    """Classe abstraite pour la structure de données Dictionary"""

    def lookup(self, key):
        """Cherche la valeur correspondante à key.
        Retourne None si key ne se trouve pas
        dans le dictionnaire"""

    def bind(self, key, value):
        """Ajoute la paire (key, value) au dictionnaire.
        Si une paire de la forme (key, x) se trouve déjà dans le dictionnaire,
        alors la valeur de key sera modifiée"""

    def contains(self, key):
        """Vérifie s'il y a une paire (key, value) dans le dictionnaire"""

    def delete(self, key):
        """Élimine toute paire de la forme (key, value) du dictionnaire"""

    def keys(self):
        """Retourne la liste de toutes les clés dans le dictionnaire"""

    def size(self):
        """Retourne le nombre de clés dans le dictionnaire"""
```

Pour ce faire, on suivra de près l'exemple des listes présenté en cours.

Exercice 1. On commencera pour écrire une classe `Noeud`, analogue à la classe `MaillotDeListe` utilisée pour les listes. Voici une première esquisse de ce que pourrait être cette classe :

```
class Node():
    """Classe pour représenter un noeud
    à l'intérieur d'un arbre de recherche binaire"""

    def __init__(self, key, value=None):
        self.key=key
        self.content=value
        self.leftNode=None
        self.rightNode=None
        self.parent=None

    def link_left(self, other):
        """L'autre sera relié à gauche se self"""

    def link_right(self, other):
        """L'autre sera relié à gauche se self"""

    def set_root(self):
        """On ne veut pas un candidat parent"""

    def set_leaf(self):
        """On ne veut pas des candidtas enfants"""
```

Exercice 2. On continuera en écrivant un premier morceaux de la classe `BinarySearchTree` :

```
4  class BinarySearchTree(Dictionary):
5
6      def __init__(self,key,value = None):
7          self.root = Node(key, value)
8          # Initialisez eventuellement dáutres attributs dóbjet
9
10     def __getNodeByKey(self, key):
11         """Recherche d'un noeud dans arbre en suivant le critère pour les
12         arbres binaire de recherche. Il retourne le noeud.  Si la clé
13         n'est pas trouvé, on retourne None, et on stocke dans des
14         attributs d'objet le noeud en bas duquel il faut éventuellement insérer
15         la clés recherchée et la direction."""
```

La fonction `__getNodeByKey` est l'analogue de `__getMaillot` pour les listes. Donc elle interviendra dans la plupart des autres fonctions.

Exercice 3. Maintenant, en utilisant `__getNodeByKey`, vous pouvez aisément implémenter ces trois fonctions :

```
10 class Dictionary():


13     def lookup(self, key):
14         """Cherche la valeur correspondante à key.
15         Retourne None si key ne se trouve pas
16         dans le dictionnaire"""
17
18     def bind(self, key, value):
19         """Ajoute la paire (key, value) au dictionnaire.
20         Si une paire de la forme (key, x) se trouve déjà dans le dictionnaire,
21         alors la valeur de key sera modifiée"""
22
23     def contains(self, key):
24         """Vérifie s'il y a une paire (key, value) dans le dictionnaire"""
```

Parmi les trois, la plus difficile est `bind`. Rappelons comment elle fonctionne :

- on demande à `__getNodeByKey` le noeud qui contient la clé ;
- si un noeud est retourné, alors on change la valeur du contenu du noeud ;
- si `None` est retourné, on trouvera dans des attributs de l'objet, le noeud et la direction ou il faut ajouter un nouvel noeud. Écrivez donc le code pour ajouter ce nouvel noeud à la bonne position.

Exercice 4. Afin de faire efficacement des tests, ajoutez à votre classe la méthode `__str__` (ou une méthode `print`) qui vous permettra de visualiser la structure de l'arbre. Par exemple, on pourra faire un parcours en profondeur de l'arbre en affichant à fur et à mesure les clés (et les valeurs).
Vous étés maintenant en mesure de faire de petit tests ; faites-les.

Exercice 5. Implémentez maintenant la méthode,

```
10 class Dictionary():


26     def delete(self, key):
27         """Élimine toute paire de la forme (key, value) du dictionnaire"""
```

Pour ce faire, vous aurez besoin d'implémenter une méthode `__getMinKeyNode` qui permet de rechercher le noeud avec la clé la plus petite en bas d'un noeud donné.
Rappelons le fonctionnement de la methode `delete(key)` :

- on cherche le noeud n qui contient la clé `key` ;
- si ce noeud à au plus un enfant, alors on peut tout de suite remplacer ce noeud par son seul fils, ou le détruire si il n'a pas de fils ;
- si le noeud n a deux enfants, il faut chercher le noeud m (avec clé valuer c et v) du fils à droite avec clé minmale ; on reaffecte au noeud n le couple (c, v) et on remplace/supprime le noed m.

Exercice 6. Terminez. Les autres méthodes

```
class Dictionary():


    def keys(self):
        """Retourne la liste de toutes les clés dans le dictionnaire"""

    def size(self):
        """Retourne le nombre de clés dans le dictionnaire"""
```

ne devraient pas poser des problèmes.
Pour `keys` on pourra faire un parcours de l'arbre. Pour `size` on pourra ajouter un attribut d'objet qui contient la taille de l'arbre, à mettre à jour à chaque ajout et à chaque rimotion d'une clé.

Exercice 7. Vous vous rappelez, depuis le TP 3, du script `dictionaryTest.py` ? Utilisez-le pour tester la classe `BinarySearchTree`.