Polytech Nice-Sophia | Examen d'ASD | **Année 2017-2018**
SI 3 | *Durée : 1 heure. Aucun document autorisé.* | **mercredi 28 mars**

*Vous devez écrire toutes les méthodes demandées dans le **cadre strict** des classes données en annexe. En particulier, vous ne devez définir ou utiliser **aucune** autre méthode que celles demandées. Le barême est indicatif et suceptible d'être ajusté. Durant toute l'épreuve, il ne sera répondu à **aucune** question.*

1 Pile avec min et max (4 points)

Cet exercice consiste à définir la classe `StackMinMax` qui implémente des *piles* offrant les méthodes classiques (`push`, `pop`, `peek` et `len`) plus deux nouvelles méthodes `peekmin` et `peekmax`. Ces deux nouvelles méthodes retournent respectivement l'élément *minimum* et l'élément *maximum* actuellement dans la pile, sans toutefois les retirer. Bien entendu, ce type de pile ne contient que des éléments *comparables*. De plus, toutes les méthodes de cette classe doivent avoir une complexité en $\Theta(1)$. Pour simplifier le code, on peut supposer que les paramètres sont toujours corrects. Complétez les méthodes ci-dessous :

```
class StackMinMax:

    def __init__(self): # le constructeur
        self.stackmin = []
        self.stackmax = []
        self.stack = []

    def __len__(self): # Retourne le nombre d'éléments dans la pile


    def peek(self): # Retourne le sommet de la pile sans le dépiler




    def peekmin(self): # Retourne l'élément minimum actuellement dans la pile




    def peekmax(self): # Retourne l'élément maximum actuellement dans la pile
```

(suite de la classe `StackMinMax`*)*

```
    def push(self,x): # Empile x dans la pile
```

```
    def pop(self): # Retire le sommet de la pile et le retourne
```

2 Propriété d'arbres binaires (7 points)

Dans cet exercice, on dispose de la classe `Tree` qui implémente des arbres binaires sans propriété particulière :

```
class Tree :

        def __init__(self,data=None,left=None,right=None) :
             self.data, self.left, self.right = data, left, right

        @property
        def data(self): return self.data

        def __getitem__(self,i) : return self.right if i else self.left
```

Etant donné un arbre binaire contenant des éléments *comparables*, on veut tester si cet arbre est un *Arbre Binaire de Recherche* (*ABR* dans la suite). On rappelle qu'un arbre binaire est un *ABR* si et seulement il est vide, ou bien si ces sous-arbres gauche et droit sont des *ABR* et si la racine de l'arbre est supérieure à tous les éléments du sous-arbre gauche et inférieure à tous les éléments du sous-arbre droit.

(suite de l'exercice 2)

Question 2.1 (2 points) *Ecrivez les fonctions* `minimum` *et* `maximum` *qui respectivement retournent l'élément minimum et l'élément maximum d'un arbre binaire* ***quelconque*** *(et non* ***pas*** *d'un ABR) :*

```
def minimum(tree):
```

```
def maximum(tree):
```

Question 2.2 (2 points) *En utilisant les deux fonctions précédentes, complétez la fonction* `is_bst_1` *qui teste si un arbre binaire est un ABR. Cette fonction retourne* `True` *si l'arbre est un ABR,* `False` *sinon :*

```
def is_bst_1(tree):
```

(suite de l'exercice 2)

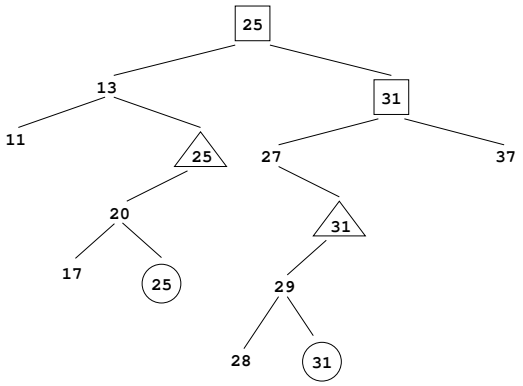
Question 2.3 (1 point) *Expliquez brièvement et concisément pourquoi la complexité de la fonction précédente n'est pas optimale :*

Question 2.4 (2 points) *Ecrivez les fonctions* `aux` *et* `is_bst_2` *telles que* `is_bst_2` *ait un comportement identique à* `is_bst_1` *mais avec une complexité linéaire. Autrement dit, la complexité de* `is_bst_2(T)` *est en* $\Theta(|T|)$ *où* $|T|$ *est le nombre d'éléments de* T *:*

```
def is_bst_2(tree):
```

3 Arbres Binaires de Recherche à Occurences (4 points)

Un *Arbre Binaire de Recherche à Occurences* (ou *ABRO* dans la suite) est un arbre binaire de recherche pouvant contenir plusieurs éléments identiques. Etant donné un *ABRO* A, les différents éléments x identiques s'appellent les *occurences* de x. Les occurences d'un élément donné sont implicitement ordonnés suivant l'ordre dans lequel on les a ajouté dans l'arbre. Ainsi, la première occurence de x dans A est l'élément ajouté en premier dans A parmi tous les éléments égaux à x dans A. Plus généralement, la $i^{\text{ème}}$ occurence de x dans A est le $i^{\text{ème}}$ élément égal à x ajouté dans A parmi tous les éléments égaux à x de A. Quand on ajoute un élément x dans un *ABRO* A dont la racine contient l'élément x, le nouvel élément x est ajouté dans le sous-arbre gauche de A. Par exemple, l'ajout successif des éléments $\boxed{25}$, 13, $\boxed{31}$, 11, $\triangle 25$, 27, 37, $\triangle 31$, 29, 28, $\bigcirc 31$, 20, $\bigcirc 25$ et 17 dans un *ABRO* initialement vide donne l'*ABRO* suivant :



Dans l'arbre ci-dessus,  est la première occurence de 25,  est la deuxième occurence de 25 et  est la troisième occurence de 25 dans l'arbre. De même, les éléments  ,  et  sont respectivement la première, deuxième et troisième occurence de 31.

Etant donnée la classe `BST` dont voici un extrait

```
class BST :
     class Node :
          def __init__(self,data) : ....
          def __getitem__(self,i) : ....
          def __setitem__(self,i,v) : ....

     def __init__(self) :
          self.__root = None

     @property
     def root(self) : return self.__root

     def is_empty(self) : return self.root is None

     def contains(self,value) :
          node = self.root
          while node is not None :
               if value == node.data : return True
               node=node[value>node.data]
          return False
```

(suite de la classe `BST`*)*

```
def insert(self,value) :
     if self.root is None :
          self.root = self.__class__.Node(value)
          return True
     node = self.root
     while True :
          if value == node.data : return False
          cond = value > node.data
          if node[cond] is None :
               node[cond] = self.__class__.Node(value)
               return True
          node = node[cond]
```

on veut la modifier pour que les arbres de cette class soient des *ABRO*. Pour celà, il faut modifier les méthodes `contains` et `insert` comme suit :

- `contains` : cette méthode prend maintenant un nouveau paramètre `n` (un entier) tel que `t.contains(x,n)` retourne `True` si `t` contient au moins `n` occurences de `x`, ou bien `False` si `t` contient moins de `n` occurences `x`

- `insert` : cette méthode ne retourne maintenant plus rien car il est toujours possible d'ajouter un élément dans l'arbre, même s'il est déjà présent, et les occurrences d'un élément sont toujours ajoutées dans le sous-arbre gauche

Complétez les nouvelles méthodes `contains` et `insert` :

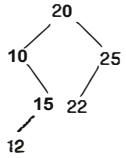
```
def contains(self,value,n=1) :
```

(suite de l'exercice 3)

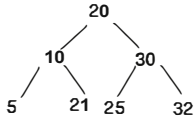
```
def insert(self,value)
```

4 Arbre AVL (6 points)

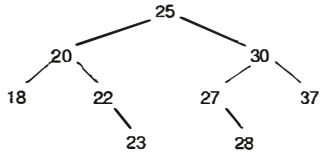
Question 4.1 (2 points) *Cochez parmi les arbres suivants ceux qui sont des AVL.*



○



○



○

○

Question 4.2 (1 point) *Quel est le nombre* ***maximum*** *d'éléments d'un AVL de hauteur h ? Expliquez !*

Question 4.3 (1 point) *Quel est le nombre* ***minimum*** *d'éléments d'un AVL de hauteur* ***5*** *? Expliquez !*

Question 4.4 (2 points) *Dessinez l'arbre AVL qu'on obtient si on ajoute successivement les entiers 9, 4, 1, 3, 2, 8, 10, 6, 5, 11 et 7 dans un arbre AVL initialement vide (dessinez les différents AVL intermédiaires jusqu'à l'AVL final)*