Feuille de TP n° 3 : programmation objet, piles, files, ABR

Exercice 1. ***Programmation objet***

a) Récupérer le fichier `classes_tp3.py` sur Moodle et étudier la définition des classes en Python.

Initialisation

Pour chaque classe, on définit une fonction `__init__(self)`. Il s'agit d'une fonction d'initialisation exécutée lors de la création d'un nouvel objet (une nouvelle instance) de la classe.

Par exemple, lorsqu'on écrit `L=list()`, on crée une nouvelle instance de la classe `list` qui est initialisée par une fonction `__init__(self)`.

Autre exemple, l'appel `s=Stack()` crée une nouvelle instance de la classe Stack initialisée par :

```
def __init__(self):
    self.size = 0
    self.items = list()
```

Mot clé `self`

Le mot clé `self` désigne ici l'instance qui est créée.

Dans le fichier, la fonction `push(self,x)` est aussi définie dans la classe du fait de l'indentation.

`self` désigne l'instance sur laquelle on exécutera cette fonction, ce mot clé permet d'appeler la fonction par : `s.push(5)`.

Lors de cette appel, `self` référence `s`, et `x=5`.

Une telle fonction est nommée méthode. Toute méthode s'appelle en écrivant quelque chose de la forme `mon_objet.ma_methode(param)`, où `param` désigne les valeurs des paramètres qui suivent `self` dans la définition de `ma_methode`.

Héritage

La classe `BST` est définie par

```
class BST(Tree):
    pass  # à remplacer par du contenu futur
```

Dans cette définition, on écrit que `BST` est une classe qui hérite (ou dérive) de la classe `Tree`. Pour l'instant, on n'a pas encore mis de contenu dans cette classe. Comme il est obligatoire d'écrire quelque chose dans une définition de classe, on utilise temporairement le mot clé `pass` qui désigne une instruction vide.

`pass` peut être utilisé dans d'autres contextes.

Concrètement, un objet créé par `t=BST()` aura accès à toutes les méthodes de la classe `Tree`, en particulier, en absence d'une méthode `__init__` dans la classe `BST`, il est initialisé comme un `Tree`.

b) Ecrire une classe `Queue` de la même manière que `Stack` et utilisant les méthodes `pop(0)` et `append` des listes Python.

Exercice 2. ***ABR***

a) A l'aide des classes `Stack` et `Queue` de l'exercice 1, définir dans la classe `Tree` deux méthodes itératives `depth_first_print(self)` et `breadth_first_print(self)`.

NB : noter que la classe `BST` hérite de la classe `Tree`.

Dans les tests, lorsqu'on crée une instance `t` de `BST`, `t` est aussi un `Tree` par héritage, donc on peut utiliser toutes les méthodes de la classe `Tree`, par exemple `infix_print`.

b) Définir dans la classe `BST`, une méthode `search(self,x)`. Si la donnée `x` est présente, `search` renvoie la référence au noeud qui a cette valeur, sinon elle renvoie `None`.

c) Définir dans la classe `BST`, une méthode `min(self)`. Si l'arbre est vide, elle renvoie `None`, sinon la référence au noeud qui a la valeur minimum.

d) Définir dans la classe `BST`, une méthode `insert(self,x)`. Elle effectue l'insertion de la valeur `x` dans l'ABR.

e) Définir dans la classe `BST`, une méthode `suppress(self,x)`. Elle effectue le retrait de la valeur `x` dans l'ABR. Elle renvoie une référence au noeud retiré si il existe, `None` sinon (dans le cas où `x` n'était pas dans l'arbre).

f) Définir dans la classe `BST`, une méthode `successeur(self,x)`. Elle renvoie la référence au noeud qui a la plus petite valeur de l'arbre supérieure strictement à `x`, si cette valeur n'existe pas, elle renvoie `None`.

g) Comparer par des tests, le temps d'exécution de n insertions d'entiers aléatoires dans un ABR initialement vide avec un tri rapide sur un tableau de n entiers.

Exercice 3. ***Piles et files***

Les implémentations des piles et files de l'exercice 1 ne permettent pas de faire les opérations push, enqueue et dequeue à temps constant.

a) Ré-écrire les classes Stack et Queue en utilisant un tableau de taille constante.

b) Ré-écrire les classes Stack et Queue en utilisant des listes chainées.