# TP2

October 23, 2015

## 1 Structures de données

### 1.1 Set et tables de hachage

On va regarder un peu la structure *set* de python et comparer ses performances avec la *list*. Un *set* est une structure qui représente un **ensemble**. C'est-à-dire que chaque valeur ne peut y être stockée qu'une seule fois. On le définit comme ça :

```
In [ ]: s = {1,4,3,5}
        print s
```

ou bien...

```
In [ ]: s = set()
        s.add(1)
        s.add(4)
        s.add(3)
        s.add(5)
        print s
```

Il est aussi possible de créer un set à partir d'une liste.

```
In [ ]: l = [2,4,2,3,3,4]
        s = set(l)
        print s
```

Pour tester si une valeur appartient à un set, on utilise la syntaxe suivante, similaire à celle des listes.

```
In [ ]: 2 in s
```

```
In [ ]: 5 in s
```

Ci-dessous, la fonction du TP précédent qui tire un tableau aléatoire.

```
In [ ]: def random_tableau(n):
            import random
            return [random.randint(1,n) for i in xrange(n)]
```

Tirez un tableau de taille 100 et construisez le *set* correspondant.

```
In [ ]:
```

Comparez les performances de la recherche de valeurs en utilisant les commandes python "v in l" et "v in s". Puis faites de nouvelles comparaisons avec des tailles plus grandes.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

## 1.2 Listes chainées

Ci-dessous un objet minimal pour représenter les listes chaînées en python

```
In [ ]: class Cellule:

            def __init__(self, valeur):
                self.valeur = valeur
                self.suivante = None

            def __repr__(self):
                return str(self.valeur)

        class Liste:

            def __init__(self):
                self.premiere = None

            def __repr__(self):
                s = "["
                c = self.premiere
                while not c is None:
                    s += str(c)
                    if c.suivante is not None:
                        s+=" -> "
                    c = c.suivante
                s+="]"
                return s
```

Vous n'avez pas besoin de comprendre entièrement ce code, ni même de maîtriser la programmation objet pour la suite du TP. Voilà comment vous pouvez l'utiliser.

Création d'une cellule :

```
In [ ]: c1 = Cellule(4)
        c2 = Cellule(8)
        print c1
        print c2
```

Accès aux différents champs :

```
In [ ]: print c1.valeur
```

```
In [ ]: c1.suivante = c2
```

```
In [ ]: c2.suivante is None
```

Création de la liste

```
In [ ]: L = Liste()
        print L
```

Comme j'ai accroché c2 à c1, il me suffit de modifier *L.premiere* pour obtenir la liste contenant c1 puis c2.

```
In [ ]: L.premiere = c1
        print L
```

Complétez les fonctions suivantes de façon à ce qu'elles correspondent à la documentation et passent les tests.

In [ ]:
```
"""
Ajoute une cellule contenant la valeur v en tête de liste
Input :
    - L, une liste chainée
    - v, une valeur
La fonction doit créer la cellule contenant la valeur v et l'ajouter en tête de liste.
"""
def ajoutTete(L,v):
    # écrire le code
```

In [ ]:
```
LTest = Liste()
ajoutTete(LTest,3)
ajoutTete(LTest,2)
ajoutTete(LTest,1)
assert(str(LTest)=="[1 -> 2 -> 3]")
```

In [ ]:
```
"""
Supprime la cellule en tete de liste
Input :
    - L, une liste chainée
Output : la valeur de la cellule supprimée. Si la liste est vide, on renvoie None et on ne fait
"""
def supprimeTete(L):
    # écrire le code
```

In [ ]:
```
LTest = Liste()
ajoutTete(LTest,3)
ajoutTete(LTest,2)
ajoutTete(LTest,1)
assert(supprimeTete(LTest)==1)
assert(supprimeTete(LTest)==2)
assert(supprimeTete(LTest)==3)
assert(supprimeTete(LTest) is None)
assert(str(LTest)=="[]")
```

In [ ]:
```
"""
Transforme une liste classique python en liste chaînée
Input :
    - une liste python
Output: la liste chaînée correspondante
"""
def fromListToListe(l):
    # écrire le code
```

In [ ]:
```
assert(str(fromListfoListe([1,2,3,4]))=="[1 -> 2 -> 3 -> 4]")
assert(str(fromListfoListe([]))=="[]")
assert(str(fromListfoListe([5,2,4,1,3]))=="[5 -> 2 -> 4 -> 1 -> 3]")
```

In [ ]:
```
"""
Supprime la premiere occurence de v dans L
Input :
    - L, une liste chainée
    - v, une valeur
```

```
        Output : True si une valeur a été supprimée et False sinon
        """
        def supprimeListe(L,v):
            # écrire le code
```

In [ ]:
```
LTest = Liste()
ajoutTete(LTest,3)
ajoutTete(LTest,2)
ajoutTete(LTest,1)
ajoutTete(LTest,2)
assert(supprimeListe(LTest,1))
assert(supprimeListe(LTest,2))
assert(supprimeListe(LTest,3))
assert(not supprimeListe(LTest,1))
assert(supprimeListe(LTest,2))
assert(str(LTest)=="[]")
```

In [ ]:
```
"""
Ajoute une cellule contenant la valeur v dans une liste triée
Input :
    - L, une liste chainée dont les valeurs sont triées par ordre croissant
    - v, une valeur
La fonction doit créer la cellule contenant la valeur v et l'ajouter à la bonne position dans L
"""
def ajoutTrie(L,v):
    # écrire le code
```

In [ ]:
```
LTest = Liste()
ajoutTrie(LTest,6)
ajoutTrie(LTest,3)
ajoutTrie(LTest,10)
ajoutTrie(LTest,4)
ajoutTrie(LTest,4)
ajoutTrie(LTest,1)
assert(str(LTest)=="[1 -> 3 -> 4 -> 4 -> 6 -> 10]")
```

In [ ]:

A présent qu'on sait insérer dans une liste triée. On peut implanter un algorithme de tri sur les listes : il suffit de récupérer les cellules une à une puis de insérer dans une nouvelle liste.

In [ ]:
```
"""
Trie la liste L donnée en entrée
Input :
    - L, une liste chainée
"""
def TrieListe(L):
    # écrire le code
```

In [ ]:
```
LTest = Liste()
ajoutTete(LTest,3)
ajoutTete(LTest,2)
ajoutTete(LTest,1)
ajoutTete(LTest,2)
TrieListe(LTest)
assert(str(LTest) == "[1 -> 2 -> 2 -> 3]")
```

Quelle est la complexité de cet algorithme ? (double cliquez sur la question pour écrire votre réponse)

Par ailleurs, la façon dont nous avons écrit les fonctions fait qu'à chaque suppression, on ne récupère que la valeur et pas la cellule elle-même. On doit donc re-créer des cellules ce qui couteux en temps et en mémoire. **Sauf si vous aviez déjà corrigé ce problème, modifiez votre fonction pour éviter la perte / re-création de cellule**. Par exemple, vous pouvez écrire de nouvelles versions de *ajoutTete* et *supprimeTete* qui travaille sur les cellules et non plus les valeurs.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

On cherche à faire la fusion de deux listes chaînées triées pour obtenir une troisième liste elle-même triée. Par exemple si on a :

```
In [ ]: L1 = Liste()
        ajoutTrie(L1,2)
        ajoutTrie(L1,4)
        ajoutTrie(L1,3)
        ajoutTrie(L1,1)
        L1
```

```
In [ ]: L2 = Liste()
        ajoutTrie(L2,5)
        ajoutTrie(L2,2)
        ajoutTrie(L2,3)
        L2
```

Alors, la liste fusionnée devra être [1 -> 2 -> 2 -> 3 -> 3 -> 4 -> 5].

La fonction prendra deux listes en paramètres et retournera la liste résultat. Les cellules seront supprimées au fur et à mesure pour être ajoutées à la liste résultat. Donc à la fin, les listes L1 et L2 seront vides !

**Réfléchissez à un algortihme, puis implantez-le. Ecrivez aussi des tests vérifiant le résultat sur plusieurs exemples à l'aide de *assert* .**

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```