TP 5 - Les Listes Chaînées

L. JOANNIC

Septembre - octobre 2024

Les objets manipulés dans ce TP seront des `Sommets`, dont les seuls attributs sont respectivement

- le nom (`s.nom` est le nom du Sommet s),
- l'altitude (`s.altitude` est l'altitude du Sommet s, en m),
- le massif (`s.massif` est le nom du massif contenant le Sommet s).

Ils sont définis dans le fichier `Sommets.py` qu'il convient donc d'importer au besoin dans l'en-tête des fichiers de travail.

Les `Sommets` sont comparables entre eux, en utilisant simplement les symboles mathématiques usuels (s1 $<$ s2 : s1 est inférieur à s2).

On peut afficher un sommet `s` en saisissant simplement `print(s)`.

```
from Sommets import *
s=Sommet('Chamechaude', 2062, 'Chartreuse')
print(s)
```

Le résultat de l'affichage est le suivant.

```
Chamechaude                          [ Chartreuse ] :      2 062 m
```

Dans un premier temps, on utilisera le fichier `Chartreuse.csv`.

On peut visualiser son contenu :

```
with open('Chartreuse.csv') as src:
    ligne = src.readline()
    while ligne:
        nom, altitude = ligne.rstrip().split(',')
        print('{:35s}{:>10s}'.format(nom, altitude))
        ligne=src.readline()
```

```
Grande Sure (La)                     1 920 m
Rocher de Lorzier (Le)               1 838 m
Rochers de Chalves (Les)             1 845 m
Rocher de l'Église (Le)              1 300 m
Pointe de la Gorgeat (La)            1 486 m
Mont Joigny (Le)                     1 556 m
Mont Outheran (Le)                   1 673 m
```

```
Cochette (La)                          1 618 m
Roc de Gleisin (Le)                    1 434 m
Roche Veyrand (La)                     1 429 m
Dent de l'Ours (La)                    1 820 m
Petit Som (Le)                         1 772 m
Grand Som (Le)                         2 026 m
Charmant Som (Le)                      1 867 m
Pinéa (La)                             1 771 m
Néron (Le)                             1 299 m
Mont Granier (Le)                      1 933 m
Sommet du Pinet (Le) ou le Truc        1 867 m
Grand Manti (Le)                       1 818 m
Scia (La)                              1 791 m
Lances de Malissard (Les)              2 045 m
Dôme de Bellefont (Le)                 1 975 m
Dent de Crolles (La)                   2 062 m
Piton de Bellefont (Le)                1 958 m
Chamechaude                            2 082 m
Grands Crêts (Les)                     1 489 m
Mont Saint-Eynard (Le)                 1 379 m
Écoutoux (L')                          1 406 m
Rachais (Le)                           1 050 m
```

1re partie : Manipulation depuis l'interface fournie

Dans cette partie, on prendra soin de coder les fonctionnalités demandées dans un fichier `TP5_1.py`, dans lequel on importera la librairie `LSC.py` fournie, ainsi que le fichier `Sommets.py`.

Exploitation.

La bibliothèque `LSC.py` jointe à ce TP fournit une implémentation du Type Abstait `Liste_Simplement_Cha` vu en cours.

Elle met, donc, à disposition une structure de données dont les primitives sont

- `creer_liste_vide()` : nil $\rightsquigarrow$ LSC, renvoyant une LSC vide.
- `est_vide(liste)` : LSC $\rightsquigarrow$ bool, Vrai ssi la liste est vide.
- `ajouter_en_tete(liste, element)` : LSC $\times$ objet $\rightsquigarrow$ LSC, renvoie une nouvelle liste constituée en abondant la liste fournie de l'élément.
- `tete(liste)` : renvoie l'élément en tête de liste, sans le supprimer.
- `queue(liste)` : renvoie la sous-liste constituée à partir de la liste initiale en supprimant la tête.

Afin de simplifier le travail d'étude, une fonction `afficher_liste(liste)` est aussi implémentée, qui permet un affichage du contenu des LSC.

```
from Sommets import *
from LSC import *
```

```
liste_sommets = creer_liste_vide()

liste_sommets = ajouter_en_tete(liste_sommets,
                                Sommet('Chamechaude', 2062, 'Chartreuse'))
liste_sommets = ajouter_en_tete(liste_sommets,
                                Sommet('Mont Blanc', 4807, 'Mont Blanc'))

afficher_liste(liste_sommets)
```

L'affichage est alors le suivant.

```
+-----
|
+- Mont Blanc                          [ Mont Blanc ] :      4 807 m
|
+- Chamechaude                         [ Chartreuse ] :      2 062 m
|
+-----
```

1. Coder une fonction `csv2liste` qui prend en paramètre le fichier `csv` des sommets d'un massif et en construit la liste.

   Afficher le résultat obtenu.

   APPELER LE PROFESSEUR POUR VERIFICATION

2. (a) Écrire la fonction `copier_liste(liste)` qui renvoie une copie du contenu de la `liste`.

   (b) Afficher le résultat obtenu. Que constate-t-on ?
   APPELER LE PROFESSEUR POUR VERIFICATION

   (c) Modifier alors cette fonction pour que la copie soit fidèle. (On pourra proposer 2 variantes, l'une itérative, l'autre récursive).

Avant de passer à la suite, insérer plusieurs occurences des mêmes sommets dans un ordre aléatoire et effectuer plusieurs copies de la liste obtenue, afin de disposer de jeux de test.

1. Coder une fonction `rechercher(liste, nom)` qui répond à la question "le sommet dont le nom est donné est-il dans la liste ?"

2. Coder une fonction `modifier_altitude(liste, nom)` qui permet de corriger l'altitude d'un sommet de nom donné (attention aux spécifications).

   APPELER LE PROFESSEUR POUR VERIFICATION

3. Coder une fonction `supprimer_sommet(liste, nom)` qui permet de supprimer la première occurence dans la liste du sommet de nom donné.

4. Coder une fonction `supprimer_sommets(liste, nom)` qui permet de supprimer toutes les occurences dans la liste du sommet de nom donné.

Indexation

On souhaite pouvoir disposer d'une indexation (entière) des éléments de nos LSC.
À cette fin, coder les fonctions suivantes.

1. `longueur(liste)` : renvoie le nombre d'éléments de la LSC.

   Proposer d'abord une approche séquentielle, puis une approche récursive.

   APPELER LE PROFESSEUR POUR VERIFICATION

2. `insérer(liste, element, rang)` : renvoie une copie de la liste initiale dans laquelle l'élément fourni est inséré au rang fourni.

   (Quelle condition faut-il vérifier sur le rang ?)

   APPELER LE PROFESSEUR POUR VERIFICATION

3. `supprimer(liste, rang)` : renvoie une copie de la liste initiale dans laquelle l'élément de rang fourni est supprimé.

   (Même question quant aux préconditions.)

4. `modifier(liste, rang, element)` : renvoie une copie de la liste initiale dans laquelle l'élément fourni remplace l'élément de rang fourni.

   (Toujours la même question.)

5. `rechercher(liste, element)` : renvoie le rang de l'élément fourni dans la liste, -1 s'il est absent.

6. `lire(liste, rang)` : renvoie l'élément de rang fourni dans la liste.

7. `trier(liste, ordre)` : renvoie une copie de la liste fournie, triée dans l'ordre fourni.

2nde partie : Implémentations.

Dans cette partie, on réalisera une librairie par implémentation.
Attention, dans la mesure du possible, on proposera une solution fonctionnelle, **sans effet de bord**.

À partir des `list` Python

En premier lieu, il est nécessaire de remplacer l'import de la librairie LSC par l'import de `list_implemente_LSC`, dans laquelle on développera les codes de cette sous-partie.

1. Implémenter la structure de liste simplement chainée à partir du type `list` de Python3.

   On développera pour cela l'interface

   - `creer_liste_vide()`
   - `est_vide(liste)`
   - `tete(liste)`
   - `queue(liste)`
   - `ajouter(liste, element)`

Un soin particulier doit être apporté à l'axiomatique, notamment des pré-conditions, et à leurs vérifications.

APPELER LE PROFESSEUR POUR VERIFICATION

2. Tester les fonctionnalités de la première partie.

   Les fonctionnalités développées sur les listes ne dépendent pas de l'implémentation retenue. Celle-ci est transparente.

   En particulier, on peut choisir de changer cette implémentation sans avoir à modifier aucune des fonctionnalités développées sur l'interface.

3. Pour aller plus loin:

   (a) Quel est le coût algorithmique de fonctionnalités `ajouter()`, `supprimer()`, `modifier()`, `rechercher()` ?

   (b) Proposer une nouvelle implémentation dans le cas d'un ajout en queue, en observant les différences éventuelles de coût.

À partir des tuples

En premier lieu, il est nécessaire de remplacer l'import de la librairie `list_implemente_LSC` par l'import de `tuple_implemente_LSC`, dans laquelle on développera les codes de cette sous-partie.

On donne l'implémentation suivante (fournie dans `tuple_implemente_LSC`):

```
def creer_liste_vide():
    return ()

def est_vide(liste):
    return (len(liste)==0)

def ajouter(liste, element):
    return (element, liste)

def tete(liste):
    resultat, _ = liste
    return resultat

def queue(liste):
    _, resultat = liste
    return resultat
```

1. Simuler, sur papier, la séquence d'instructions suivante, en recopiant et en complétant la table de suivi de la variable `liste`.

instruction	contenu de liste	affichage
`liste = creer_liste_vide()`		
`ajouter(liste, 1)`		
`ajouter(liste, 2)`		
`ajouter(liste, 3)`		
`tete(liste)`		
`liste = queue(liste)`		
`liste = queue(liste)`		

2. On ajoute la fonctionnalité suivante.

```
def changer_tete(liste, element):
    """
    Remplace la valeur de tete de la liste par element.
    """
    liste[0] = element
    return None
```

   (a) Tester l'instruction `changer_tete(liste, 0)`.
   Que se passe-t-il ?

   (b) Quelle difficulté est-elle mise en évidence ?

   (c) Proposer une nouvelle version de `changer_tete` qui lève la difficulté.

   APPELER LE PROFESSEUR POUR VERIFICATION

3. Vérifier la cohérence des codes précédemment développés.