Chapitre 7

Tableaux et implémentation des listes en python

7.1 Introduction

• Les objets de la classe `<list>` (les "listes") constituent une **structure de données séquentielle** qui s'avère essentielle en `python`.

De leur implémentation dépend le coût de leurs diverses méthodes :

1. Quel est le coût de l'accès à un élément par son indice ?
2. Quel est le coût de l'ajout d'un élément à une liste ?
3. Quel est le coût de la suppression d'un élément d'une liste ?
4. Quel est le coût de la fonction `len()` qui retourne la longueur d'une liste passée en argument ? etc...

• En parlant de *coût* ce sont la complexité en temps (dans le pire, le meilleur des cas, ...) et la complexité en espace (occupation de la mémoire dans le pire, le meilleur des cas) qui nous intéressent.

• Les complexités d'algorithmes implémentant des listes en dépendent.

• C'est la **structure de donnée théorique** utilisée pour l'implémentation des lites qui détermine ces coûts.

• Et qu'en est-il des tableaux `numpy` ?

7.2 Structure de données : les tableaux

7.2.1 Notion de tableaux

• En informatique théorique un **tableau** est une structure de données séquentielle, pour des données de même type.

• Les données d'un tableau sont contenues en mémoire de façon contiguë. Lors de la création d'un tableau de taille N, un espace contiguë de N fois la taille nécessaire pour stocker une donnée du type considéré est réservé en mémoire.

éléments	elt1	elt2	elt3	...	eltN
indices	0	1	2	...	N-1

• L'accès à un élément du tableau (lecture ou écriture) se fait à l'aide de son indice en temps $O(1)$ (il suffit d'ajouter $(i-1)$ fois la taille du type considéré à l'adresse du premier élément du tableau pour accéder au i-ème élément).

• On ne peut pas insérer dans un tableau un nouvel élément, ou y supprimer un élément.

• La longueur d'un tableau est mémorisée lors de sa création. On l'obtient ainsi en temps $O(1)$.

7.2.2 Les tableaux de `numpy`

• C'est cette structure de donnée qu'utilise un tableau `numpy` :

1. Un tableau `numpy` est un tableau (statique) de donnée homogènes. Par défaut le type des données est `float`.
2. On peut utiliser durant la création tout autre type de donnée numérique, par l'option `dtype = 'int'` (pour des entiers signés), `dtype = 'uint'` (pour des entiers non signés), etc...

```
In[1] : import numpy as np
In[2] : T = np.zeros(4)
In[3] : T
Out[3] : array([0.0, 0.0, 0.0, 0.0, 0.0])
In[4] : T = np.zeros(4, dtype = 'int')
In[5] : T
Out[5] : array([0, 0, 0, 0, 0])
```

3. La taille d'un tableau `numpy` est fixée à la création.
   Ajout, suppression d'éléments est impossible.

7.2.3 Coût des opérations sur les tableaux `numpy`

• Complexité des opérations sur les tableaux `numpy` :

Opération	Complexité
Création d'un tableau de N éléments	$O(N)$
Lecture d'un élément	$O(1)$
Modification d'un élément	$O(1)$
Obtention du nombre d'éléments	$O(1)$
Ajout d'un élément	Impossible
Suppression d'un élément	Impossible

• Les tableaux `numpy` ne peuvent contenir que des données numériques de même type.

• C'est la structure de donnée la plus efficiente lorsque l'on manipule des tableaux de données numérique de taille et de type fixés à l'avance.

7.2.4 Structure de données : les tableaux dynamiques

• En informatique théorique un **tableau dynamique** est une structure de données séquentielle, pour des données de même type, redimensionnable.

• Les données d'un tableau sont contenues en mémoire de façon contiguë. Un espace contiguë de N fois la taille nécessaire est réservé en mémoire. Les éléments du tableau sont stockés en début de tableau ; leur nombre est la longueur n du tableau. Capacité N et longueur n sont stockées en mémoire.

éléments	elt1	elt2	elt3	...	elt_n	-	...	-
indices	0	1	2	...	n-1	n	...	N-1

• L'accès à un élément du tableau (lecture ou écriture) se fait à l'aide de son indice en temps $O(1)$. La longueur d'un tableau s'obtient en temps $O(1)$.

• Suppression d'un élément en temps $O(n)$ dans le pire des cas (les élément suivants doivent être décalés d'un rang à gauche). L'élément d'indice i est supprimé en temps $O(n-i)$.

• Tant que la capacité n'est pas dépassée on peut insérer dans un tableau un nouvel élément en temps $O(n)$ dans le pire des cas. L'ajout d'un élément à l'indice i se fait en temps $O(n-i)$.

• Si la capacité maximale est atteinte il faut copier dans un nouveau tableau, avec un coût $O(N)$.

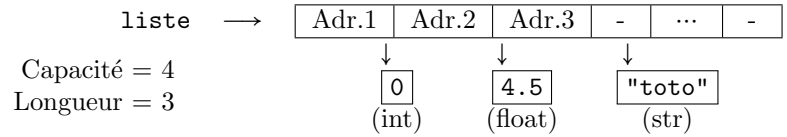
7.2.5 Implémentation des listes en `python`

• En `python` une liste est implémentée comme un tableau dynamique.

• A ceci près que dans une liste `python` les données ne sont pas homogènes (ne sont pas forcément du même type). Ce ne sont pas les données qui sont stockées dans le tableau, mais des pointeurs, c'est à dire les adresses mémoires des emplacements où sont stockées ces données, de façon à obtenir des tableaux homogènes.

• Lorsque le nombre d'éléments approche la capacité, la taille du tableau est augmentée, si nécessaire par copie sur un nouvel emplacement.

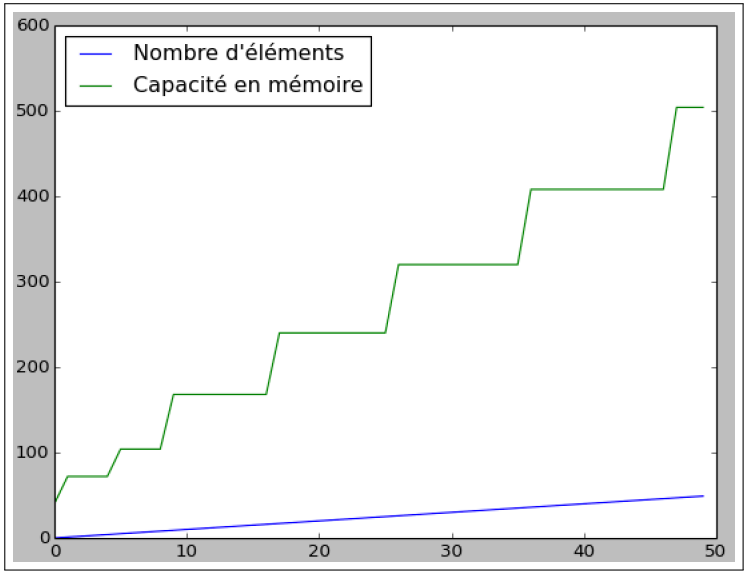
Exemple : l'implémentation de `liste=[0, 4.5, "toto"]` ressemble à :



7.2.6 Mise en évidence

On augmente le nombre d'éléments d'une liste `L`,
on mémorise dans 2 listes `X` et `Y` ses nombre d'éléments et capacité,

```
L = []
X,Y=[],[]
for k in range(50):
    X.append(len(L))          # Nbre d'éléments
    Y.append(L.__sizeof__())    # capacité
    L.append(1)
import matplotlib.pyplot as plt
plt.plot(X,X)
plt.plot(X,Y)
plt.legend(("Nombre d'éléments","Capacité en
mémoire"),'upper left')
```



Observons les changements de capacité :

A chaque changement de capacité on écrit le couple :
(Nbre d'éléments, Nouvelle capacité).

```
L = []
a = 0
for k in range(1000):
    b = L.__sizeof__()      # capacité
    if b != a:
        print("(",len(L),",",b,")",sep='',end=';')
        a = b
    L.append(1)
```

```
(0,40);(1,72);(5,104);(9,168);(17,240);(26,320);(36,408);
(47,504); (59,616);(73,744);(89,888);(107,1048);(127,1224);
(149,1424); (174,1648);(202,1904);(234,2192);(270,2512);
(310,2872);(355,3280); (406,3736);(463,4248);(527,4824);
(599,5472);(680,6208);(772,7032); (875,7960);(991,9000);
```

7.2.7 Coût des opérations sur les listes en `python`

• Ainsi en `python` pour une liste de longueur n :

1. L'accès à un élément d'une liste par son indice est de complexité $O(1)$.
2. La longueur de la liste s'obtient en $O(1)$.
3. La suppression d'un élément est dans le pire des cas en $O(n)$. La suppression du dernier élément est en $O(1)$.
4. L'ajout d'un élément est dans le pire des cas en $O(n)$. L'ajout d'un élément en fin de liste (avec `append()`) est dans le pire des cas en $O(n)$; on dit (informellement pour nous !) que c'est en O(1) "'*en temps amorti*" (le coût d'une éventuelle duplication sera "amortie" par les $n-1$ ajouts suivants...)
5. ... dans le pire des cas, ajouter n éléments, ou plus généralement $a.n+b$ éléments, avec a, b des constantes, est aussi dans le pire des cas de complexité $O(n)$.

Pour un calcul de complexité d'un algorithme utilisant des listes, mieux vaut réserver au départ une liste de capacité suffisante et éviter l'usage de `.append`.

• Complexité des opérations sur les listes de `python` :

Opération	Complexité
Création d'une liste de n éléments	$O(n)$
Lecture d'un élément	$O(1)$
Modification d'un élément	$O(1)$
Obtention du nombre d'éléments n	$O(1)$
Suppression de l'élément d'indice i	$O(n-i)$
Suppression du dernier élément (`.pop()`)	$O(1)$
Ajout d'un élément à l'indice i	$O(n-i)$ si $n < N$ $O(n)$ si $n = N$
Ajout d'un élément en fin de liste (`.append()`)	$O(1)$ si $n < N$ $O(n)$ si $n = N$ $O(1)$ en temps amorti

• Les "listes" forment une structure de donnée hybride propre à `python`. Très souple d'utilisation, à utiliser lorsque l'on manipule des tableaux dynamiques ou de données non numériques ou non homogènes.

• Ce n'est pas ce que l'on entend communément par "liste" en Informatique.