NSI Terminale - Structure de données

La structure de données liste

qkzk

2020/03/31

La structure de donnée liste

Objectifs

Notre objet d'étude aujourd'hui est la structure de données linéaire **liste**.

Les objectifs de ce travail sont :

- de *définir* la structure de données liste. Pour cela nous allons nous concentrer sur ses méthodes,
- de *manipuler* cette structure de données,
- d'appréhender la notion de *mutabilité* des listes (elles peuvent changer),
- d'appréhender la *complexité* de la manipulation des listes,
- de comprendre que ce qui est appelé `List` *en Python n'est pas une liste au sens commun du terme*.

La structure de donnée

Vous connaissez déjà la structure de liste puisque vous l'avez largement utilisée dans les programmes Python que vous avez pu écrire précédemment. Vous avez créé des listes, ajouté des éléments, accédé à sa longueur, accédé à un élément, etc.

Néanmoins vous ne vous êtes jamais interrogés sur ce qu'était la liste en tant que structure de données.

Qu'est-ce qu'une liste ?

- Intuitivement. Une liste est une collection finie d'éléments qui se suivent. C'est donc une structure de données *linéaire*.
- Une liste peut contenir un nombre quelconque d'éléments y compris nul (la liste vide).

Obtenir une définition formelle

Prenons une liste comme par exemple ‘$\ell_1 = [3, 1, 4]$‘. C’est une liste à trois éléments (ou de longueur trois) dont le premier est ‘3‘, le deuxième ‘1‘, et le dernier ‘4‘.

Une façon de décrire cette liste consiste à dire que

- la liste ‘ℓ_1‘ possède un premier élément ‘3‘ qu’on nommera élément de *tête*,
- et que vient après cet élément de tête la liste ‘$\ell_2 = [1, 4]$‘ des éléments qui suivent, liste qu’on nommera *reste*.

Ce qu'on vient de dire de la liste 'ℓ_1' peut être répété pour la liste 'ℓ_2' qui est donc constituée :

- d'un élément de *tête* : '1',
- et d'un *reste* : '$\ell_3 = [4]$'.

À nouveau on peut répéter le même discours pour la liste 'ℓ_3' qui est donc constituée :

- d'un élément de *tête* : '4',
- et d'un *reste* : '$\ell_4 = []$'.

La liste 'ℓ_4' étant vide, elle ne possède pas d'élément de tête, et ne peut donc pas être décomposée comme nous venons de le faire à trois reprises.

Si on convient d'utiliser la notation '(x, ℓ)' pour désigner le couple constitué de l'élément 'x' de tête, et du reste 'ℓ' d'une liste, on peut alors écrire :

$$`\ell_1 = (3, (1, (4, [\,])))`$$

Ce qui vient d'être fait pour la liste 'ℓ_1' peut être reproduit pour n'importe quelle liste.

On peut conclure cette approche en donnant une définition abstraite et formelle des listes d'éléments appartenant tous à un ensemble 'E'.

Une *liste* d'éléments d'un ensemble 'E' est

- soit la liste vide
- soit un couple '(x, ℓ)' constitué d'un élément '$x \in E$' et d'une liste 'ℓ' d'éléments de 'E'.

Les listes peuvent donc être vues comme *des structures de données récursives*.

Primitives sur les listes

Primitives ?

Primitives Les *opérations primitives* d'une structure de donnée sont les opérations minimales qui permettent de définir la structure et de lui donner les méthodes attendues.

Primitives sur les listes

Constructeur. Une liste est soit la liste vide,

soit un couple constitué de l'élément de tête suivi de la liste des éléments qui suivent.

Le constructeur de liste doit donc permettre de produire soit une liste vide et pour cela aucun argument n'est nécessaire, soit une liste à partir de deux arguments.

Primitives sur les listes

Sélecteurs. Les listes non vides possèdent une tête et un reste.

Il nous faut les sélecteurs pour accéder à ces deux composantes.

Prédicat Un prédicat testant si une liste est vide est utile.

Prédicat *: une question dont la réponse est un booléen (V/F)*

Résumé des primitives

Une structure de donnée **liste** doit implémenter :

1. **La construction** à partir d'une liste vide ou à partir d'un couple tête (élément) et reste (liste).
2. **La sélection** de l'élément de tête ou du reste.
3. **Prédicat.** on doit pouvoir répondre à la question : "cette liste est-elle vide ?"

Liste vs tableaux

Qu'est ce qui différencie les listes des tableaux ?

Tableau

Sa taille est *fixe !*
Les éléments *se suivent en mémoire*.
Accéder à un élément par son indice est rapide

Liste

Les éléments ne se suivent pas forcément en mémoire.
La queue la liste *pointe vers une autre liste*
Accéder à un élément par son indice est lent (il faut suivre tous les liens)

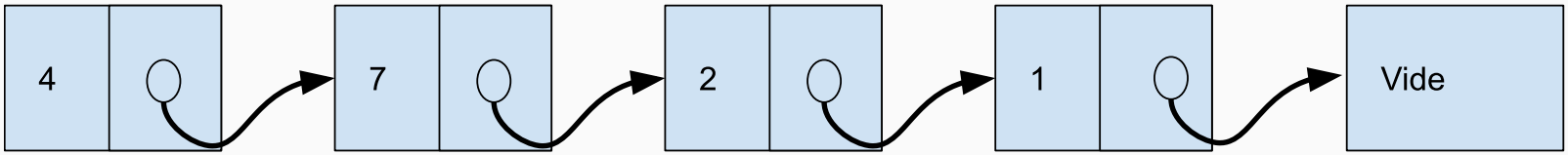
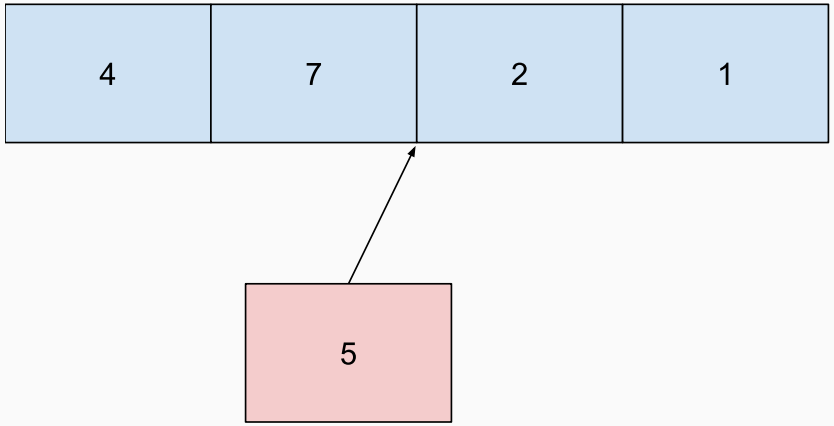


Figure 1: liste_vs_tableau

Exemple d'opération : insérer un élément dans un tableau

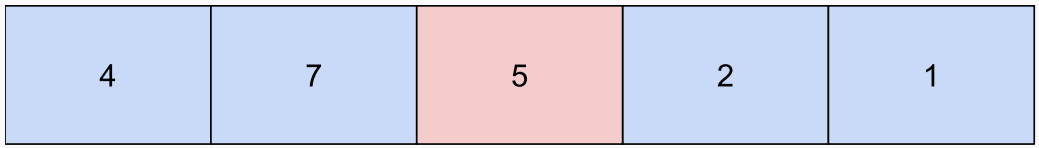
Tableau : taille 4

Pour **insérer** un élément il faut *recréer un nouveau tableau de taille supérieure !*



Nouveau tableau : taille 5

Dans lequel on **recopie** tous les éléments
C'est très lent !



Exemple d'opération : insérer un élément dans une liste

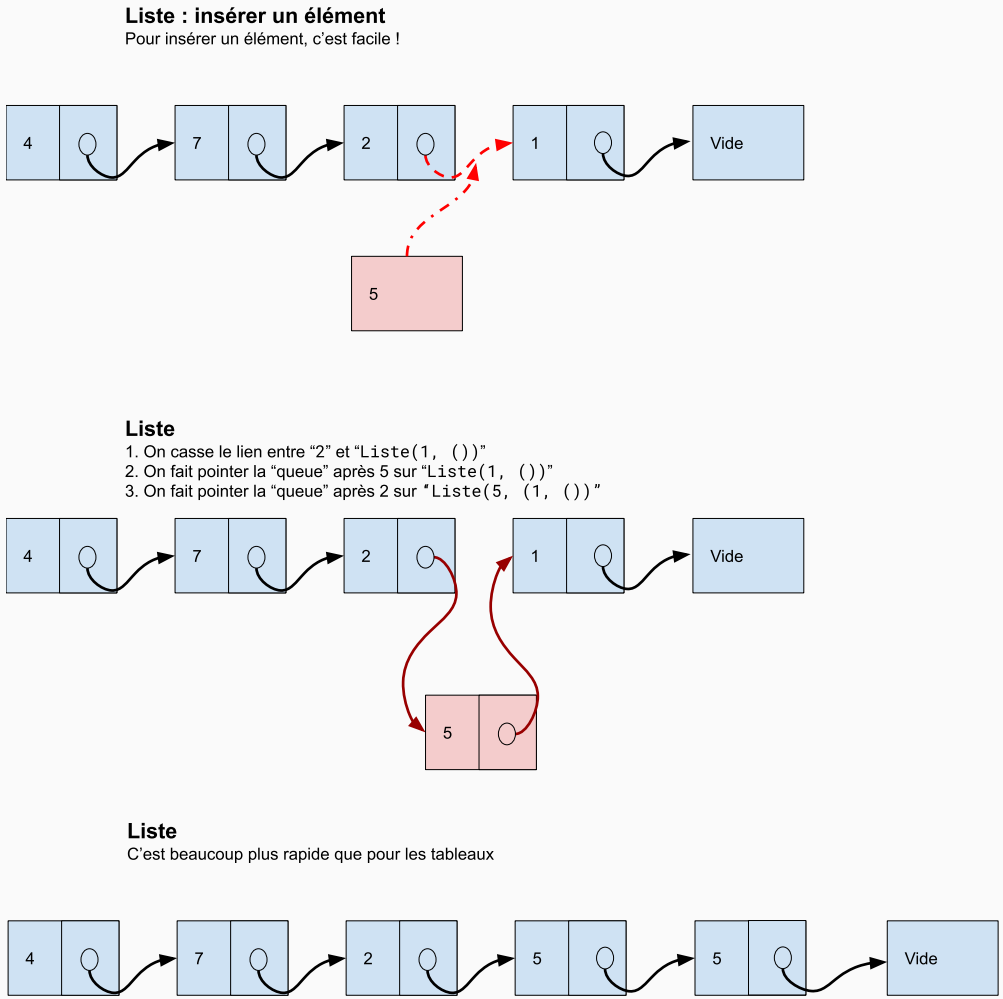


Figure 3: liste_ajouter_element

Et en Python ?

En python les objets `List` ne sont ni des tableaux, ni des listes. Ce sont des **tableaux dynamiques** (`array` en javascript ou autre).

Ils combinent les avantages des deux mais aussi une partie des inconvénients.

La philosophie de Python est de viser la simplicité *d'usage* avant l'efficacité optimale. Si *vraiment* la vitesse est un facteur important, il convient d'utiliser d'autres outils que les `List` Python pour certaines opérations.

Complexité

Coût des opérations

Pourquoi implémenter plusieurs structures ? Après tout, on peut *tout faire* avec des listes Python !

Parce que l'efficacité est fondamentale. Certaines structures sont plus adaptées à certains problèmes.

Comparaison des coûts d'opération

Accéder à un élément : tableau

Pour accéder à l'élément 2 du tableau

```
T = ['a', 'b', 'c']
```

1. On se rend à l'adresse où débute `T`
2. On se déplace de deux positions. On lit : `'c'`

Le temps est constant : Accéder se fait en complexité $O(1)$.

Accéder à un élément : liste

Pour accéder à l'élément 2 de la liste

```
L = ('a', ('b', ('c', [])))
```

1. On se rend à l'adresse où débute `L`
2. On suit le lien jusque l'adresse de la queue du premier élément
3. On suit le lien jusque l'adresse de la queue du second élément
4. On lit la valeur de la tête : `'c'`

Le temps est linéaire : Accéder se fait en complexité $O(n)$.

Modifier un élément

Comme on l'a vu plus haut, c'est le contraire !

Cette opération est plus rapide pour les listes que pour les tableaux.

Une classe `Liste` minimale en Python

Implémenter une structure minimale

Pour implémenter une structure de donnée il faut :

1. avoir décrit les primitives
2. décrire les méthodes qui seront à construire plus tard
3. Utilise la programmation objet pour créer un nouveau *type* de données

Construction de la liste

Voici comment **construire** :

```
class Liste: # avec un e !!!
  def __init__(self, *args):
    if len(args) == 0:
      self.__contenu = None
    else:
      self.__contenu = {
        "tete": args[0],
        "queue": args[1]
      }
```

Un exemple :

```
>>> l = Liste() # liste vide : l = []
>>> g = Liste(1, l) # tête : 1, queue : []
>>> h = Liste(2, g) # tête : 2, queue : (1, [])
```

Sélecteurs

```
# toujours dans la classe List

def tete(self):
  if self.__contenu is None:
    return None
  else:
    return self.__contenu["tete"]

def queue(self):
  if self.__contenu is None:
    return None
  else:
    return self.__contenu["queue"]
```

Sélecteurs, un exemple

```
>>> g.tete()
1
>>> h.queue()
<__main__.Liste object at 0x7f66fbdecf70>
>>> isinstance(h, Liste)
True # c'est bien un objet de type liste
```

Prédicat : *est vide*

```
  # toujours dans la classe liste
  def est_vide(self):
    if self.__content is None:
      return True
    else:
      return False
```

Exemple

```
>>> h.est_vide()
False
>>> l.est_vide()
True
```

Méthodes utiles

Notre structure de donnée est incomplète

Cette structure de donnée que nous avons défini ne contient par toutes les méthodes des `List` en Python. Imaginons avoir :

```
l = ['a', 'b', 'c']
```

- **longueur** :

```
>>> len(l)
3
```

- **sélection** d'un élément quelconque

```
>>> l[2]
'c'
```

- **mutabilité** :
  - modifier un élément :

```
>>> l[2] = 'd'
```

  - supprimer un élément

```
>>> del l[1]
>>> l
['a', 'b']
```

  - position d'un élément :

```
>>> l.index('b')
1
```

  - ajout d'un élément :

```
>>> l.append('e')
>>> l
['a', 'b', 'e']
```

- **Présenter la liste**

```
>>> l # python affiche le contenu de ses objets
['a', 'b', 'e']
>>> g = Liste()
>>> g # nous obtenons un résultat peu pratique...
<__main__.Liste object at 0x7f66fadccf978>
```

TP compléter la classe liste

Durant un prochain TP nous allons tenter d'implémenter le plus simplement possible ces méthodes.

Nous aurons ainsi l'occasion de travailler :

- la programmation objet,
- la récursion (les listes sont naturellement récursives !)
- la structure de donnée elle-même (on se limitera aux méthodes déjà crées !)